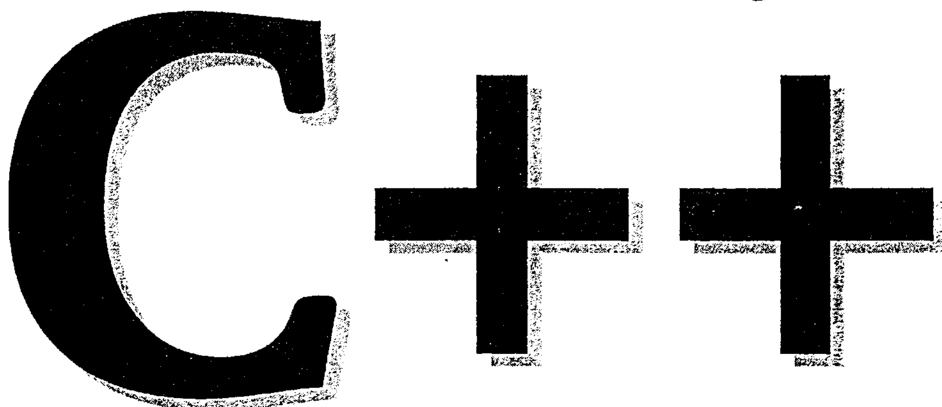
Kris Jamsa



Manualul începătorului

Traducere de Lucian Limona

Teora

Titlul original: Rescued by C++, Third Edition

Traducerea din limba engleză s-a făcut după ediția originală publicată în Statele Unite ale Americii în anul 1997.

Distribuție

București: B-dul Al. I. Cuza nr. 39; tel./fax: 222.45.33
Sibiu: Șos. Alba Iulia nr. 40; tel.: 069/21.04.72; fax: 069/23.51.27

Teora – Cartea prin poștă

CP 79-30, cod 72450 București, România
Tel./Fax: 252.14.31
e-mail: cpp@teora.kappa.ro

Teora

CP 79-30, cod 72450 București, România
Fax: 210.38.28
e-mail: teora@teora.kappa.ro

Copertă: Gheorghe Popescu
Tehnoredactare: Techno Media
Șef de redacție: Dana Stanciu
Director general: Teodor RĂDUCANU

NOT 3120 CAL C++, MANUAL INCEPATOR
ISBN 973-20-0139-9
Printed in Romania

Cuprins

PARTEA I

Elemente de bază

În cadrul acestei părți veți învăța elementele de bază care vă sunt necesare pentru a putea crea propriile dumneavoastră programe în C++. Dacă nu ați mai scris niciodată un program nu este cazul să vă îngrijorați, deoarece în această parte vom face împreună primii pași. Pentru a porni la lucru va trebui să instalați compilatorul *Turbo C++ Lite*, produs de Borland și aflat pe CD-ROM-ul care însoțește această carte. Cu ajutorul compilatorului *Turbo C++ Lite* veți putea să creați, să compilați, să rulați și să depanați (să înlăturați erorile din) propriile dumneavoastră programe C++. La vremea când veți fi parcurs lecțiile simple prezentate în această parte, veți fi deja pe propriile picioare în ceea ce privește programarea în C++! Lecțiile conținute în această parte sunt următoarele:

Lecția 1

Instalarea compilatorului Borland C++ Lite

Așa cum veți afla pe parcursul celei de a doua lecții, „Crearea primului program“, pentru a programa în C++ trebuie să plasați instrucțiunile programului într-un fișier sursă ASCII. Apoi, prin intermediul unui program special care se numește *compilator*, veți transforma instrucțiunile programului C++ pe care le puteți înțelege dumneavoastră în seriile de unu și de zero pe care le poate înțelege și executa calculatorul. În prezent, programatorii profesioniști cumpără și folosesc compilatoare C++ precum Borland C++ sau Microsoft *Visual C++*. Pentru a vă ajuta să porniți la drum cu programarea în C++, firma Borland International a permis editurii Teora să includă pe CD-ROM-ul care însoțește această carte compilatorul *Turbo C++ Lite*. După cum veți vedea, compilatorul *Turbo C++ Lite* este un program destinat mediului MS-DOS. În cazul în care utilizați Windows, veți putea rula compilatorul *Turbo C++ Lite* în cadrul unei ferestre DOS. În această lecție veți învăța să instalați și să utilizați compilatorul *Turbo C++ Lite*. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Compilatorul *Turbo C++ Lite* este un compilator destinat mediului MS-DOS.
- Prin intermediul compilatorului *Turbo C++ Lite* puteți să creați și să editați propriile programe de instrucțiuni C++, cu posibilitatea de a salva aceste instrucțiuni într-un fișier sursă pe hard-disc.
- După ce creați propriul fișier sursă C++, puteți utiliza compilatorul *Turbo C++ Lite* pentru a compila instrucțiunile programului în seriile de unu și zero pe care calculatorul le poate rula.
- Cu ajutorul compilatorului *Turbo C++ Lite* puteți compila cea mai mare parte a programelor prezentate în această carte.
- Spre deosebire de alte compilatoare C++, cum ar fi Borland C++ sau Microsoft *Visual C++*, atunci când compilați un program cu *Turbo C++ Lite*, programul creat de către compilator nu poate fi executat decât din mediul *Turbo C++ Lite*.
- Pentru a crea un fișier executabil pe care să-l poată rula și alți utilizatori, va trebui să compilați programul cu ajutorul unor compilatoare precum Borland C++ sau Microsoft *Visual C++*.

Instalarea compilatorului Turbo C++ Lite

Așa cum am amintit, *Turbo C++ Lite* este un compilator destinat mediului MS-DOS. În cazul în care utilizați Windows, puteți să rulați compilatorul *Turbo C++ Lite* în cadrul unei ferestre DOS, așa cum se vede în Figura 1.1

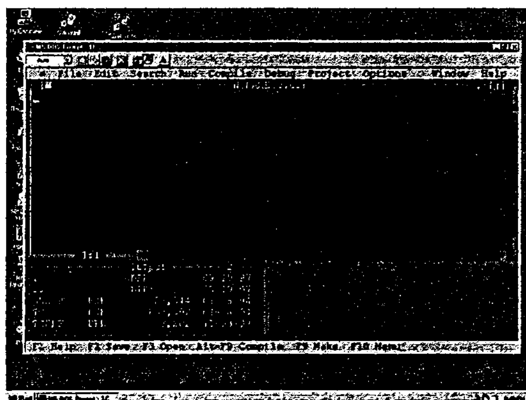


Figura 1.1 *Rularea compilatorului* ***Turbo C++ Lite*** *în cadrul unei ferestre DOS.*

Pentru a instala compilatorul *Turbo C++ Lite* din mediul Windows, parcurgeți următorii pași:

1. Introduceți CD-ROM-ul care însoțește cartea în unitatea de CD-ROM a calculatorului.
2. Dacă utilizați Windows 95, selectați comanda Run din meniul Start. Dacă utilizați Windows 3.1, selectați comanda Run a meniului File din Program Manager. În replică, Windows va afișa caseta de dialog Run, ilustrată în Figura 1.2:

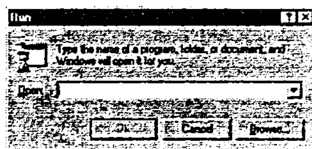


Figura 1.2 *Caseta de dialog Run.*

3. În cadrul casetei de dialog Run tastați numele programului de instalare, *D:\SETUP*, înlocuind litera de unitate D cu acea literă care corespunde unității dumneavoastră de CD-ROM. De exemplu, în cazul în care CD-ROM-ul dumneavoastră reprezintă unitatea E, tastați *E:\SETUP*. Programul de instalare va afișa caseta de dialog Destination Path, înfățișată în Figura 1.3.
4. Caseta de dialog Destination Path vă permite precizarea directorului de pe hard-disc în care va fi instalat compilatorul *Turbo C++ Lite*. În mod implicit, fișierele vor fi instalate în directorul *TCLITE*. Dacă optați pentru directorul *TCLITE*, selectați OK pentru a merge mai departe. Altfel, tastați calea dorită și apăsați Enter.

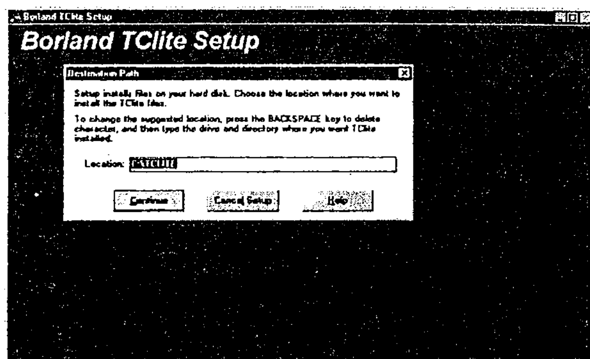


Figura 1.3 Caseta de dialog Destination Path.

Acum sunteți gata să rulați compilatorul *Turbo C++ Lite*, iar despre aceasta vom discuta în secțiunea intitulată *Rularea compilatorului Turbo C++ Lite.*

Instalarea compilatorului Turbo C++ Lite sub MS-DOS

După cum precizam, *Turbo C++ Lite* este un compilator pentru mediul MS-DOS. În cazul în care nu folosiți Windows, ci MS-DOS, urmați pașii de mai jos pentru instalarea pe hard-disc a compilatorului *Turbo C++ Lite*:

1. Introduceți CD-ROM-ul care însoțește cartea în unitatea de CD-ROM a calculatorului.
2. Pentru a copia fișierele compilatorului *Turbo C++ Lite* de pe CD-ROM pe hard-disc, folosiți următoarele comenzi *XCOPY*. Pentru fiecare dintre comenzi înlocuiți litera de unitate D cu litera ce corespunde unității dumneavoastră de CD-ROM, așa cum se vede aici:

```
C:\> XCOPY  D:\BGI\*.*         \TCLITE\BGI\*.* <Enter>
C:\> XCOPY  D:\BIN\*.*         \TCLITE\BIN\*.* <Enter>
C:\> XCOPY  D:\INCLUDE\*.*     \TCLITE\INCLUDE\*.* /S <Enter>
C:\> XCOPY  D:\LIB\*.*         \TCLITE\LIB\*.* <Enter>
```

Rularea compilatorului Turbo C++ Lite

Compilatorul *Turbo C++ Lite* este un compilator destinat mediului MS-DOS. În cazul în care utilizați Windows va trebui să deschideți o fereastră DOS pentru a rula compilatorul. Pentru deschiderea unei ferestre DOS sub Windows 95, urmați pașii de mai jos:

1. Selectați comanda Run din meniul Start. Windows va afișa caseta de dialog Run.
2. În cadrul casetei de dialog Run, tastați *COMMAND* și apăsați Enter. Windows va deschide o fereastră DOS.

Pentru a deschide o fereastră DOS sub Windows 3.1, parcurgeți următorii pași:

1. Selectați comanda Run a meniului File din Program Manager. Windows va afișa caseta de dialog Run.
2. În cadrul casetei de dialog Run, tastați *COMMAND* și apăsați Enter. Windows va deschide o fereastră DOS.

În cadrul ferestrei DOS, tastați următoarea comandă *CHDIR* pentru a selecta directorul *TCLITE\\BIN*:

```
C:\WINDOWS> CHDIR   \TCLITE\BIN <Enter>
C:\TCLITE\BIN>
```

În continuare introduceți comanda *TC* pentru a lansa compilatorul *Turbo C++ Lite*:

```
C:\TCLITE\BIN> TC    <Enter>
```

În replică, sistemul va lansa compilatorul *Turbo C++ Lite*, așa cum se poate vedea în Figura 1.4.

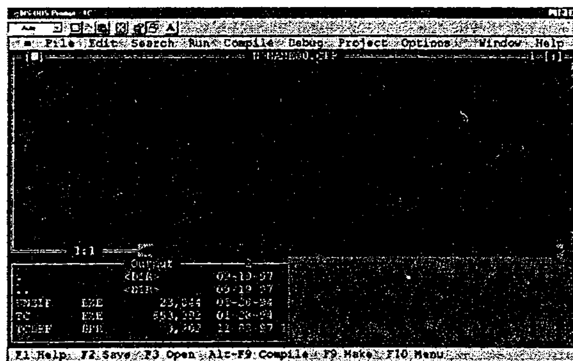


Figura 1.4 *Compilatorul* ***Turbo C++ Lite***.

Programul *Turbo C++ Lite* afișează în mod normal o fereastră de cod în cadrul căreia veți introduce instrucțiunile programului, o fereastră de rezultate în care *Turbo C++ Lite* va afișa rezultatele programului și o fereastră de mesaje în care *Turbo C++ Lite* va afișa diferite mesaje (așa cum sunt mesajele de eroare care vă anunță că ați tastat greșit o instrucțiune). Dacă ferestrele de cod, de rezultate și de mesaje nu sunt vizibile, deschideți meniul Window din *Turbo C++ Lite* și selectați comanda Tile. Dacă nici după această aranjare a ferestrelor nu puteți vedea o fereastră anume, selectați meniul Window și efectuați un clic cu mouse-ul pe comanda care corespunde ferestrei ce doriți să fie afișată.

Încărcarea unui program C++

În cea de a doua lecție veți învăța să tastați propriile programe de instrucțiuni în cadrul ferestrei de cod din Turbo C++ Lite. Pentru a economisi timp în studiul programelor prezentate de această carte, aveți posibilitatea de a încărca fișierele de programe de pe

CD-ROM-ul care însoţeşte cartea. Încărcarea unui fişier de pe CD-ROM se poate face prin intermediul comenzii Open a meniului File.

Pe CD-ROM, fişierele de programe sunt organizate după lecţii. De exemplu, programele conţinute în prima lecţie se află pe CD-ROM într-un director numit *LESSON01*. Similar, fişierele corespunzătoare celei de a doua lecţii se află într-un director numit *LESSON02*. În fine, fişierele pentru Lecţia 40 se află în directorul *LESSON40*. Spre exemplu, în directorul *LESSON01* se află fişierul de program *Demo.CPP*, care conţine următoarele instrucţiuni C++:

```
#include <iostream.h>

void main(void)
  {
    cout << "This is a sample C++ program!" << endl;
  }
```

Pentru a încărca aceste instrucţiuni în *Turbo C++ Lite*, parcurgeţi paşii următori:

1. În cadrul *Turbo C++ Lite*, selectaţi comanda Open din meniul File. *Turbo C++ Lite* va afişa caseta de dialog Load a File, aşa cum se poate vedea în Figura 1.5.

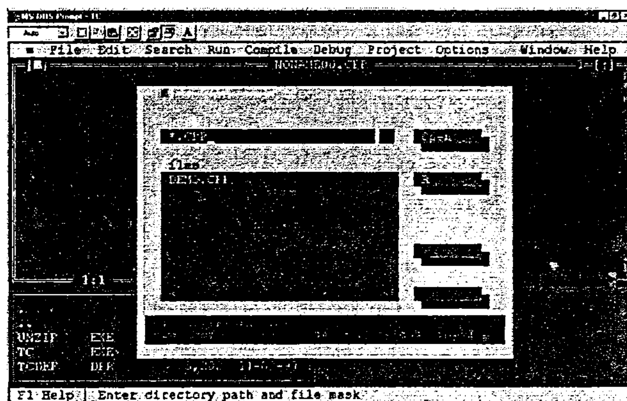


Figura 1.5 *Caseta de dialog Load a File.*

2. În cadrul casetei de dialog Load a File, introduceţi numele de fişier *D:\LESSON01\Demo.CPP*, înlocuind litera D cu litera corespunzătoare unităţii CD-ROM. *Turbo C++ Lite* va afişa instrucţiunile de program în fereastra de cod, aşa cum înfăţişează Figura 1.6.



Figura 1.6 *Încărcarea programului* ***Demo.CPP*** *în cadrul* ***Turbo C++ Lite.***

Notă: *Pentru a tipări un program, selectați comanda Print a meniului File din* ***Turbo C++ Lite.***

Rularea unui program

Atunci când programați, ceea ce faceți este să tastați instrucțiuni de program C++ pe care le veți stoca ulterior în cadrul unui *fișier sursă*. Apoi, prin intermediul unui program special numit compilator, instrucțiunile de program sunt transformate în seriile de unu și zero pe care le poate executa calculatorul. Pentru a compila un program în cadrul *Turbo C++ Lite,* selectați comanda Make EXE File din meniul Compile.

Așa cum veți afla din cea de a doua lecție, atunci când creați un program C++ trebuie să respectați reguli de sintaxă, care specifică, de exemplu, că la sfârșitul oricărei instrucțiuni trebuie plasat semnul punct și virgulă. Atunci când apare o eroare de sintaxă trebuie să editați instrucțiunile de program pentru a corecta respectiva eroare. Dacă, în schimb, instrucțiunile de program sunt corecte (nu încalcă regulile de sintaxă C++), atunci compilatorul C++ va crea un program pe care calculatorul îl va putea executa. Puteți apoi să rulați programul în cadrul *Turbo C++ Lite* selectând comanda Run a meniului Run. *Turbo C++ Lite* va afișa rezultatele programului, așa cum o arată Figura 1.7.

Fereastra de rezultate

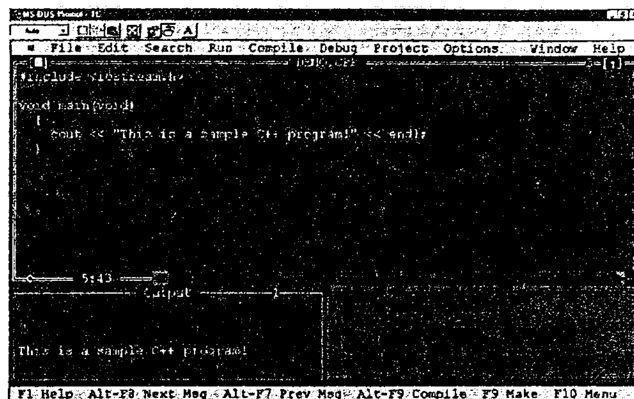


*Figura 1.7 Rezultatele programului **Demo.CPP**.*

Programele compilate cu Turbo C++ Lite pot fi rulate numai în cadrul Turbo C++ Lite, nu și de la linia de comandă

După cum ați văzut, compilatorul C++ transformă instrucțiunile de program C++ în serii de unu și zero pe care le poate executa calculatorul. Compilatorul plasează programul executabil într-un fișier *EXE*, precum *Demo.EXE*. Atunci când creați un program executabil prin intermediul *Turbo C++ Lite*, acel program nu poate fi rulat decât din mediul *Turbo C++ Lite* (cu ajutorul comenzii Run a meniului Run). Dacă încercați rularea programului de la linia de comandă, calculatorul va afișa următorul mesaj de eroare:

```
C:\TCLITE\BIN Demo <Enter>
This program can only be run
from within the IDE
```

Precum vedeți, programul *Demo* nu a afișat textul așteptat, ci un mesaj care vă informează că execuția programului trebuie făcută în cadrul mediului integrat de dezvoltare (IDE) *Turbo C++ Lite*. Dacă doriți să creați un program pe care să-l poată rula și alți utilizatori va trebui să apelați la un compilator diferit, precum Borland C++ sau Microsoft *Visual C++*.

Salvarea instrucțiunilor de program C++ într-un fișier sursă

În cadrul celei de a doua lecții veți folosi editorul *Turbo C++ Lite* pentru a tasta un program C++. După tastarea instrucțiunilor de program veți salva programul într-un fișier pe hard-disc. Pentru a efectua salvarea unui program într-un fișier, urmați pașii de mai jos:

1. Selectați comanda Save As din meniul File. *Turbo C++ Lite* va afișa caseta de dialog Save File As, ilustrată în Figura 1.8.

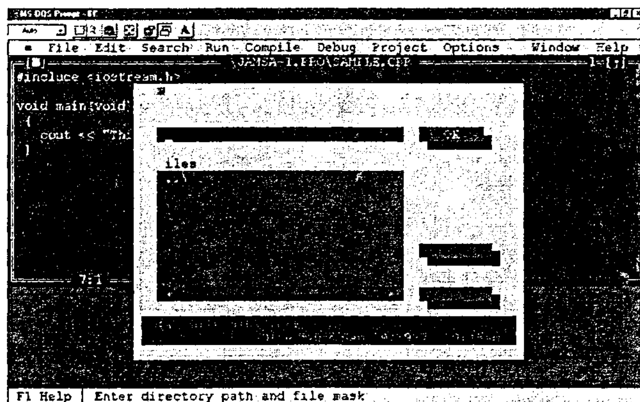


Figura 1.8 *Caseta de dialog Save As.*

2. Tastați numele de fișier dorit în cadrul casetei de dialog Save File As.
3. Selectați OK.

Pentru a vă ușura munca, CD-ROM-ul care însoțește această carte conține toate programele prezentate în lecțiile cărții. Prin intermediul comenzii Open a meniului File veți putea deschide fișiere aflate pe CD-ROM. Dacă efectuați modificări într-un program, ați putea dori să salvați acel program într-un fișier aflat pe hard-disc. Pentru a salva un fișier, selectați comanda Save As din meniul File. Apoi selectați unitatea de disc, directorul și numele fișierului de salvat în cadrul casetei de dialog Save File As.

Ce trebuie să știți

În această lecție ați învățat să instalați și să rulați compilatorul *Turbo C++ Lite*. În lecția următoare, „Crearea primului program“, veți crea mai multe programe C++. Prin intermediul compilatorului *Turbo C++ Lite* aveți posibilitatea să tastați, să compilați și să rulați programe. Dar înainte de a trece la cea de a doua lecție, asigurați-vă că ați reținut următoarele aspecte importante:

☑ CD-ROM-ul care însoțește această carte conține compilatorul *Turbo C++ Lite*, care este un compilator destinat mediului MS-DOS.

☑ Cu ajutorul compilatorului *Turbo C++ Lite* puteți să tastați programe de instrucțiuni C++, să compilați instrucțiunile de program în seriile de unu și zero pe care le poate executa calculatorul și apoi să rulați programul.

- ☑ Prin intermediul comenzii Save a meniului File din *Turbo C++ Lite* puteți să salvați programele de instrucțiuni C++ sub forma unui fișier sursă pe hard-disc.
- ☑ Atunci când creați programe executabile cu *Turbo C++ Lite*, aceste programe pot fi rulate numai din cadrul mediului *Turbo C++ Lite*.
- ☑ Pentru a crea un fișier executabil care să poată fi rulat pe orice calculator va trebui să compilați programul cu ajutorul unor compilatoare precum Borland C++ sau Microsoft *Visual C++*.

Lecția 2

Crearea primului program

Fiecare dintre noi a folosit diverse programe de calculator, precum un procesor de text, o aplicație de calcul tabelar, un browser de Web și chiar sistemul Microsoft Windows. Programele de calculator, numite și *software*, sunt fișiere care conțin instrucțiuni ce arată calculatorului cum să îndeplinească sarcini anume. De exemplu, un program de procesare de text conține instrucțiuni care arată calculatorului cum să salveze, să tipărească și chiar să verifice ortografia documentelor dumneavoastră.

Dacă lucrați în mediul Windows, de pildă, fișierele cu extensiile *EXE* și *COM* conțin comenzi executabile pe care calculatorul le poate îndeplini. Cu alte cuvinte, aceste fișiere de program conțin instrucțiuni specifice pe care calculatorul le urmează în principiu una după cealaltă pentru a îndeplini anumite sarcini. Atunci când creați un program apelați la un limbaj de programare, așa cum este C++, pentru a specifica instrucțiunile pe care vreți să le urmeze calculatorul. Apoi, prin intermediul unui program special numit compilator, instrucțiunile de program C++ sunt transformate în serii de unu și zero pe care calculatorul le poate înțelege.

În lecția de față veți învăța să formulați comenzi pentru calculator prin intermediul instrucțiunilor C++. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru a crea un program trebuie să folosiți un editor de text în scopul introducerii instrucțiunilor C++ într-un *fișier sursă* al programului.
- Pentru a transforma instrucțiunile de program C++ într-un program executabil (seriile de unu și zero pe care le înțelege calculatorul) trebuie să utilizați un program special, numit *compilator* de C++.
- Pentru a modifica sau corecta un program veți folosi un editor de text.
- Atunci când încălcați una (sau mai multe) dintre regulile de programare C++, compilatorul de C++ va afișa pe ecran mesaje de eroare de sintaxă. Atunci când compilatorul semnalează erori de sintaxă este necesar să editați programul pentru a corecta respectivele erori.
- De fiecare dată când modificați fișierul sursă C++ trebuie să compilați din nou programul pentru ca schimbările efectuate să intre în vigoare.

Programarea reprezintă procesul de definire a unei liste de instrucțiuni pe care calculatorul trebuie să le urmeze pentru a îndeplini o anumită sarcină. Pentru precizarea instrucțiunilor se folosește un *limbaj de programare*, așa cum este C++. Prin intermediul unui editor de text, instrucțiunile de program sunt plasate într-un *fișier sursă*. Apoi, cu ajutorul unui program special, un *compilator*, instrucțiunile sunt transformate din forma în care le citiți și le înțelegeți dumneavoastră în seriile de unu și zero pe care le poate înțelege calculatorul.

Cea mai bună metodă pentru a înțelege procesul de creare și compilare a unui program este chiar crearea unui program C++ simplu – exact ceea ce vom face în continuare!

Crearea unui program simplu

Primul dumneavoastră program este numit, după cum v-ați putea aștepta, *First.CPP*. Atunci când creați programe C++, folosiți extensia *CPP* pentru a indica celorlalți (programatorilor, mai ales) că fișierul dumneavoastră conține un program C++. Când veți rula mai târziu programul *First.CPP*, pe ecran va fi afișat mesajul *Rescued by C++!*. Fragmentul următor de ecran conține un prompt de linie de comandă (C:\> în exemplul nostru), comanda tastată (numele de program, *First*, urmat de Enter) și rezultatul afișat pe monitor:

```
C:\> First <Enter>
Rescued by C++!
```

Atunci când creați programe, puteți lucra într-un mediu bazat pe linie de comandă, așa cum sunt MS-DOS sau Unix, sau într-un mediu Windows. Pentru simplitate, diferitele exemple de rezultate afișate care apar în această carte presupun că lucrați în cadrul unei linii de comandă. În acest caz, pentru a executa programul *First.EXE* va trebui să tastați numele programului, *First*, și apoi să apăsați Enter.

Pentru crearea programului veți folosi un editor de text, precum EDIT (oferit împreună cu MS-DOS) sau editorul *Turbo C++ Lite* integrat în compilatorul Borland *Turbo C++ Lite* care se află pe CD-ROM-ul ce însoțește această carte, scopul fiind alcătuirea fișierului (numit fișier sursă) care conține instrucțiunile de program. Nu folosiți pentru crearea fișierului sursă al programului un procesor de text, așa cum sunt *Word®* sau *WordPerfect®*. După cum știți, procesoarele de text vă permit crearea de documente formatate, cu caractere aldine, margini aliniate și alte facilități. Pentru a formata documentele în acest fel, un procesor de text inserează în cadrul documentului anumite caractere speciale (ascunse). Asemenea caractere sunt cele care stabilesc literele cursive sau care fixează o anumită lățime a marginii. Deși aceste caractere speciale au un sens bine precizat pentru procesorul de text, ele nu vor avea semnificație în C++ și vor duce astfel la apariția de erori.

Cu ajutorul unui editor de text, tastați următoarele instrucțiuni de program C++ (fiecare caracter exact așa cum apare, folosind minuscule sau majuscule după cum se vede):

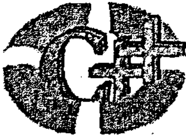
```
#include <iostream.h>

void main(void)
  {
    cout << "Rescued by C++!";
  }
```

Nu vă îngrijorați dacă instrucțiunile C++ vi se par fără sens. În cea de a treia lecție, „O privire mai atentă asupra limbajului C++", veți afla scopul fiecărei instrucțiuni. Deocamdată ar trebui să vă concentrați asupra introducerii. Aveți grijă, de exemplu, să introduceți în mod corect ghilimele, punct și virgulă și paranteze. Inspectați încă o dată cu atenție

instrucțiunile de program. Dacă totul este corect, salvați aceste instrucțiuni în fișierul *First.CPP*.

Crearea și rularea programului First.CPP în cadrul TCLITE



Așa cum ați învățat în prima lecție, „Instalarea compilatorului Borland C++ Lite", CD-ROM-ul care însoțește această carte conține și compilatorul *Turbo C++ Lite*. Pentru a crea programul *First.CPP* în cadrul *Turbo C++ Lite* parcurgeți pașii următori:

1. Lansați *Turbo C++ Lite* așa cum am discutat în prima lecție.
2. Selectați comanda New a meniului File.
3. În cadrul ferestrei de cod din *Turbo C++ Lite* tastați instrucțiunile programului *First.CPP*, așa cum o ilustrează Figura 2.1.

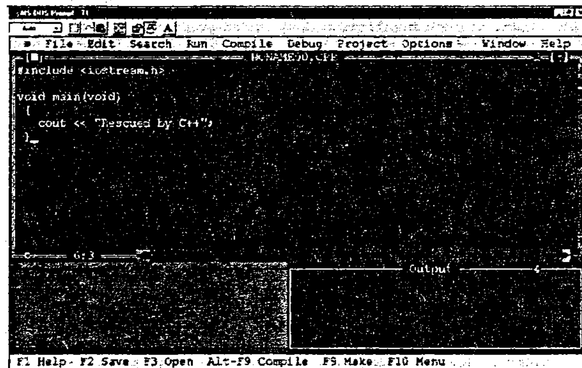


Figura 2.1 *Tastarea instrucțiunilor programului* ***First.CPP*** *în cadrul ferestrei de cod din* ***Turbo C++ Lite***.

4. Selectați comanda Save As din meniul File. *Turbo C++ Lite* va afișa caseta de dialog Save File As. Tastați numele *First.CPP* și apoi apăsați Enter.
5. Selectați comanda Run din meniul Run. *Turbo C++ Lite* va compila instrucțiunile programului, după care va rula programul, afișând rezultatele în cadrul ferestrei de rezultate din *Turbo C++ Lite*, așa cum se vede în Figura 2.2.

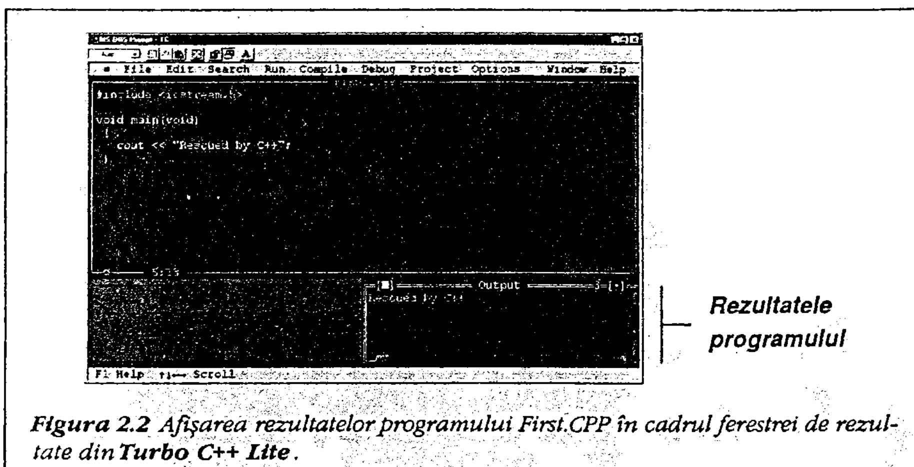


***Figura 2.2** Afişarea rezultatelor programului First.CPP în cadrul ferestrei de rezultate din **Turbo C++ Lite**.*

Atribuirea de nume sugestive pentru fişierele sursă ale programelor



Atunci când creaţi programe C++, instrucţiunile programului vor fi plasate într-un fişier sursă. La denumirea fişierului sursă al unui program folosiţi extensia *CPP* pentru a indica altor programatori (şi compilatorului de C++) că acel fişier conţine un program C++. În plus, alegeţi un nume de fişier care să sugereze scopul programului. Dacă creaţi, de pildă, un program de gestiune, aţi putea opta pentru numele *Gestiune.CPP*. Similar, în cazul unui program care calculează salariile dintr-o firmă, un nume potrivit ar fi *Salarii.CPP*. Pentru a evita confuzii, nu folosiţi niciodată ca nume de program numele unei comenzi existente în cadrul sistemului de operare, aşa cum ar fi *COPY* sau *DEL*.

Compilarea programului

Calculatoarele lucrează cu combinaţii de unu şi zero (numite *cod maşină*) care reprezintă prezenţa sau absenţa semnalelor electrice în interiorul calculatorului. Dacă un semnal are valoarea unu (este prezent), calculatorul efectuează o anumită operaţie, în timp ce dacă semnalul are valoarea zero (este absent), calculatorul poate efectua o altă operaţie. Din fericire, nu trebuie să scrieţi programe sub formă de unu şi zero (aşa cum făceau programatorii în anii '40 şi '50). Puteţi, în schimb, să scrieţi programe prin intermediul unui limbaj de programare, aşa cum este C++, după care să apelaţi la un program special, compilatorul de C++, pentru a transforma instrucţiunile programelor (codul sursă) în cod maşină.

Cu alte cuvinte, compilatorul inspectează fişierul sursă care conţine instrucţiunile de program C++. Dacă acele instrucţiuni nu încalcă nici una dintre regulile de programare C++, compilatorul va transforma instrucţiunile în cod maşină (unu şi zero) pe care

calculatorul îl poate apoi executa. Compilatorul depune codul mașină într-un fișier executabil, a cărui extensie este de obicei *EXE*. Odată ce fișierul *EXE* a fost creat, programul poate fi rulat prin tastarea numelui său în cadrul liniei de comandă și apăsarea apoi a tastei Enter.

În funcție de compilatorul pe care îl folosiți, comanda ce trebuie executată pentru utilizarea compilatorului poate varia. Pentru a afla comanda corectă de utilizare a compilatorului, apelați la documentația care îl însoțește pe acesta. După parcurgerea codului sursă al programului, compilatorul va crea un program executabil pe care îl va stoca sub forma unui fișier pe hard-disc. Dacă lucrați în mediul MS-DOS, fișierul programului executabil va avea extensia *EXE*, ca de pildă *First.EXE*.

Încărcarea de pe Internet a unui compilator de C++



Așa cum ați aflat din prima lecție, „Instalarea compilatorului Borland C++ Lite", CD-ROM-ul care însoțește această carte conține compilatorul Borland *Turbo C++ Lite*, care rulează în mediul MS-DOS. Dacă programați într-un mediu Unix sau Mac, este posibil să găsiți pe World Wide Web un compilator de C++ shareware pe care să-l puteți folosi pentru compilarea programelor prezentate în această carte. Pentru mai multe informații privind încărcarea de compilatoare pentru C++ vizitați pe Web unul dintre site-urile *http:// www.ncf.carleton.ca/~bg283/* sau *http://www.cs.princeton.edu/software/lcc/index.html.*

În cazul în care compilatorul afișează mesaje de eroare la compilarea programului, editați fișierul sursă și comparați fiecare caracter din fișier cu fiecare caracter ce apare în carte. Corectați toate greșelile, salvați modificările efectuate și apoi compilați din nou programul. După ce ați reușit compilarea programului, rulați programul prin tastarea numelui acestuia în cadrul liniei de comandă, așa cum am arătat mai devreme.

Despre compilator



La crearea unui program se folosește un limbaj de programare (cum este C++) pentru specificarea instrucțiunilor pe care calculatorul trebuie să le execute în scopul efectuării unei anumite sarcini. Cu ajutorul unui editor de text, instrucțiunile programului sunt introduse într-un fișier sursă. Apoi se folosește un program special, numit compilator, care transformă fișierul sursă în cod mașină (serii de unu și zero pe care le poate înțelege calculatorul). Dacă procesul de compilare reușește, atunci va fi creat un fișier de program executabil. Dacă, însă, introduceți greșit una sau mai multe linii sau dacă încălcați reguli C++, compilatorul va afișa pe ecran mesaje de eroare și va trebui să editați fișierul sursă pentru a îndrepta erorile cu pricina.

Dacă lucrați pe un sistem mainframe sau pe un mini-calculator, este posibil să aveți deja la dispoziție un compilator care poate fi accesat de dumneavoastră sau de către ceilalți utilizatori. Dacă folosiți un PC, atunci va trebui să cumpărați și să instalați un compilator, așa cum ar fi Borland *C++* sau Microsoft *Visual C++*.

Crearea unui al doilea program

Ați reușit să compilați și să executați programul *First.CPP*. Acum folosiți editorul de text pentru a crea un al doilea fișier, numit *Easy.CPP*, care să conțină următoarele instrucțiuni de program:

```
#include <iostream.h>

void main(void)
  {
    cout << "Programming in C++ is easy!";
  }
```

Ca mai înainte, salvați instrucțiunile de program C++ într-un fișier sursă și apoi utilizați compilatorul pentru a crea programul executabil.

Dacă procesul de compilare a programului se încheie cu succes, atunci va fi creat un program executabil numit *Easy.EXE*. La rularea programului, pe ecran va fi afișat următorul mesaj:

```
C:\> Easy <Enter>
Programming in C++ is easy!
```

În continuare, folosiți editorul pentru a edita fișierul sursă *Easy.CPP* și modificați mesajul programului pentru a include și cuvântul *very*, așa cum se vede mai jos:

```
    cout << "Programming in C++ is very easy!";
```

Salvați modificarea în cadrul fișierului sursă și compilați programul. După o compilare reușită, executați programul ca mai jos:

```
C:\> Easy <Enter>
Programming in C++ is very easy!
```

După modificarea unui fișier sursă este necesară o recompilare

De fiecare dată când modificați fișierul sursă al unui program trebuie să compilați din nou acel program pentru ca schimbările făcute să intre în vigoare. Spre exemplu, folosiți editorul de text pentru a modifica încă o dată fișierul sursă *Easy.CPP*. De data aceasta, adăugați în program o nouă instrucțiune, așa cum se vede în continuare:

```
#include <iostream.h>

void main(void)
   {
     cout << "Programming in C++ is easy!";
     cout << endl << "And pretty cool!";
   }
```

Salvați modificările în fișierul sursă. Apoi executați programul ca mai jos:

```
C:\> Easy <Enter>
Programming in C++ is easy!
```

După cum vedeți, programul nu a afișat a doua linie a mesajului. Pentru ca modificările efectuate să se reflecte în program este necesară recompilarea acestuia. Compilați, așadar, programul așa cum am discutat mai devreme și apoi rulați-l. Deoarece compilatorul a ținut seama de schimbările din codul sursă, pe ecran va fi afișată și a doua linie a mesajului, ca mai jos:

```
C:\> Easy <Enter>
Programming in C++ is easy!
And pretty cool!
```

Despre erorile de sintaxă

În orice limbaj, fie acesta engleză, franceză, germană sau chiar C++, există o serie de reguli de sintaxă care trebuie respectate în folosirea acelui limbaj. În engleză, de exemplu, propozițiile se termină de regulă cu punct, semn de exclamare sau semn de întrebare. De asemenea, la începutul unei propoziții se folosește de obicei o majusculă. Sintaxa din C++ face apel la punct și virgulă, paranteze, acolade și multe alte caractere. Atunci când omiteți sau utilizați incorect unul dintre aceste caractere, compilatorul de C++ afișează pe ecran un mesaj care descrie eroarea respectivă. În plus, majoritatea compilatoarelor vor afișa și numărul de linie din cadrul fișierului sursă al instrucțiunii cu probleme.

Compilatorul de C++ nu poate crea un program executabil până când nu sunt corectate toate erorile de sintaxă. Pentru a înțelege procesul de identificare și corectare a erorilor de sintaxă, creați următorul program, numit *Syntax.CPP*.

```
#include <iostream.h>

void main(void)
   {
     cout << Use quotes around messages;
   }
```

Dacă priviţi cu atenţie, puteţi observa că mesajele afişate de cele două programe anterioare apăreau în fişierul sursă încadrate de ghilimele. Sintaxa C++ (regulile de limbaj) impune prezenţa ghilimelelor. La compilarea programului *Syntax.CPP*, compilatorul va afişa mai multe mesaje de eroare de sintaxă. În cazul compilatorului Borland *Turbo C++ Lite* care se află pe CD-ROM-ul ce însoţeşte această carte, pe ecran vor fi afişate următoarele mesaje:

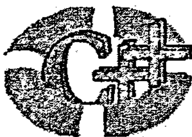
```
Compiling Syntax.CPP:
Error Syntax.CPP 5: Undefined symbol 'Use' in function main()
Error Syntax.CPP 5: Undefined symbol 'quotes' in function
                    main()
Error Syntax.CPP 5: Undefined symbol 'around' in function
                    main()
Error Syntax.CPP 5: Undefined symbol 'messages' in function
                    main()
```

În exemplul de faţă, compilatorul afişează patru erori de sintaxă. După cum puteţi vedea, toate erorile sunt în legătură cu linia a cincea a fişierului sursă. Editaţi fişierul şi încadraţi mesajul în ghilimele, ca mai jos:

```
cout << "Use quotes around messages";
```

Acum puteţi să compilaţi cu succes programul pentru crearea fişierului executabil. Atunci când sunteţi la început în folosirea unui limbaj de programare, este aproape sigur că veţi avea de corectat erori de sintaxă de fiecare dată când compilaţi un program. După câteva programe acumulate „la activ“, erorile de acest gen vor fi identificate şi corectate aproape instantaneu.

Despre erorile de sintaxă



Atunci când creaţi programe C++ trebuie să respectaţi anumite reguli. De exemplu, textul mesajelor trebuie încadrat între ghilimele, iar la sfârşitul celor mai multe instrucţiuni trebuie plasat semnul punct şi virgulă (veţi învăţa pe parcursul acestei cărţi care instrucţiuni necesită punct şi virgulă şi care nu). Dacă programul dumneavoastră încalcă o regulă de sintaxă, compilatorul de C++ va afişa pe ecran un mesaj de eroare. Pentru crearea unui program executabil de către compilator va trebui să corectaţi în prealabil toate erorile de sintaxă.

Lucrul într-un mediu Windows

Pentru simplitate, fiecare dintre exemplele anterioare se baza pe presupunerea că mediul în care lucrați este unul în stil linie de comandă, așa cum sunt MS-DOS și Unix. În prezent, însă, majoritatea programatorilor de C++ lucrează în medii care rulează sub Windows, cum ar fi *Visual C++* sau Borland *C++ 5.02* pentru Windows. Atunci când programați în cadrul unui mediu care rulează sub Windows, instrucțiunile de program rămân aceleași cu cele prezentate în exemplele din această carte. Cu alte cuvinte, instrucțiunile C++ din programul *First.CPP* sunt identice într-un mediu bazat pe Windows cu cele pe care le-ați folosi într-un mediu bazat pe linie de comandă. Ceea ce se schimbă în mediile Windows este modul de compilare și de rulare a programelor.

În cadrul unui mediu de programare sub Windows, fișierele sursă pot fi create prin intermediul unui editor integrat, iar compilarea programelor se face apoi prin selectarea unei comenzi de meniu sau prin efectuarea unui clic pe un buton aflat pe bara cu instrumente. Dacă programul conține erori de sintaxă, este posibil ca mediul de programare să afișeze mesajele de eroare într-o fereastră separată. După ce reușiți compilarea programului, rularea acestuia se face prin intermediul unei comenzi de meniu (sau al unui buton de pe bara cu instrumente). Încă o dată, este posibil ca mediul să deschidă o fereastră separată în care să afișeze derularea programului. Figura 2.3 prezintă un mediu de programare destinat sistemului Windows.

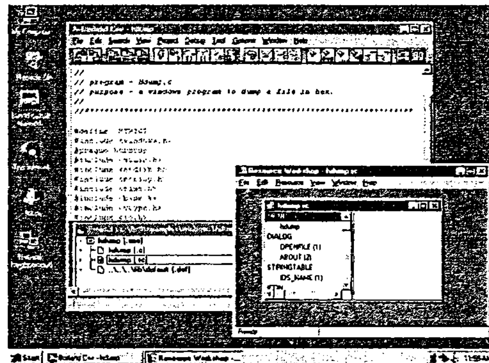


***Figura 2.3** Un mediu de programare sub Windows.*

Denumirea de mediu de programare se datorează faptului că acesta pune la dispoziție toate instrumentele necesare pentru crearea, compilarea și executarea programelor.

Ce trebuie să știți

În această lecție ați învățat să creați și să compilați programe C++. În cuprinsul celei de a treia lecții, „O privire mai atentă asupra limbajului C++", vom arunca o privire mai atentă asupra instrucțiunilor care compun programele pe care le-ați creat în lecția de față. Veți afla scopul unor cuvinte cheie precum *void*, veți vedea cum știu programele să afișeze ceva pe ecran și veți învăța că programele C++ folosesc acolade {} pentru a grupa instrucțiunile aflate într-o anumită relație. Dar înainte de a trece la cea de a treia lecție, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Programele sunt fișiere ce conțin o listă de instrucțiuni pe care calculatorul le va executa pentru a îndeplini o anumită sarcină.

☑ Pentru crearea programelor C++ se folosește un editor de text în scopul introducerii instrucțiunilor de program.

☑ Programele C++ sunt stocate în fișiere sursă a căror extensie este *CPP*.

☑ Compilatorul transformă instrucțiunile programelor C++ în serii de unu și zero (cod mașină) pe care calculatorul le poate înțelege.

☑ Asemeni oricărui limbaj, C++ are propria sa mulțime de reguli de limbaj sau sintaxă.

☑ Atunci când încălcați o regulă de sintaxă din C++, compilatorul afișează un mesaj care descrie eroarea și, eventual, numărul liniei din program care a produs acea eroare.

☑ Trebuie să înlăturați toate erorile de sintaxă pentru ca un compilator să poată crea un program executabil.

☑ După ce modificați un fișier sursă trebuie să compilați din nou programul pentru a se ține cont și de respectivele modificări.

Lecția 3

O privire mai atentă asupra limbajului C++

În cadrul lecției a doua, „Crearea primului program", ați creat mai multe programe C++. La acel moment, scopul urmărit nu era să înțelegeți instrucțiunile C++, ci mai degrabă să înțelegeți procesul de creare și compilare a programelor C++. Această lecție vă va oferi o primă privire de aproape asupra instrucțiunilor care compun un program C++. Veți vedea că majoritatea programelor C++ respectă un același format, începând cu una sau mai multe instrucțiuni *#include*, conținând mai apoi o linie *void main(void)* și continuând cu o serie de instrucțiuni grupate în program cu ajutorul acoladelor deschisă și închisă {}. Veți descoperi în lecția de față că aceste instrucțiuni care par pe undeva intimidante sunt de fapt foarte ușor de înțeles. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Instrucțiunea *#include* vă permite să profitați de *fișierele de antet* care conțin instrucțiuni C++ sau definiții.
- Partea principală a unui program C++ începe cu instrucțiunea *void main(void).*
- Programele sunt alcătuite din una sau mai multe funcții care conțin instrucțiuni înrudite al căror scop este să îndeplinească o anumită sarcină.
- Pentru a afișa ceva pe ecran, programele vor folosi de cele mai multe ori funcția de ieșire *cout.*

Atunci când creați programe C++, lucrați la nivel de *instrucțiuni.* În lecțiile care urmează veți învăța despre *instrucțiunea de atribuire*, care atribuie valori variabilelor, despre instrucțiunea *if*, care permite programului să ia decizii și așa mai departe. În continuare ne vom referi la conținutul unui program ca fiind *instrucțiunile programului.*

O privire asupra instrucțiunilor de program

În lecția a doua ați creat programul C++ numit *First.CPP* care conținea următoarele instrucțiuni:

```
#include <iostream.h>

void main(void)
  {
    cout << "Rescued by C++!";
  }
```

Programul din exemplul nostru conține trei instrucțiuni. Acoladele (numite *simboluri de grupare*) grupează instrucțiunile care se află într-o anumită relație, așa cum se vede în continuare:

```
#include <iostream.h>

void main(void)
  {
    cout << "Rescued by C++!";
  }
```

Instrucțiunile programului

Secțiunile care urmează se vor opri în detaliu asupra fiecăreia dintre instrucțiunile programului.

Despre instrucțiunea #include

Toate programele pe care le-ați creat în lecția a doua începeau cu următoarea instrucțiune *#include*:

```
#include <iostream.h>
```

Atunci când creați programe C++ aveți posibilitatea de a profita de anumite instrucțiuni și definiții pe care compilatorul vi le pune la dispoziție. La compilarea programului, instrucțiunea *#include* va determina compilatorul să includă în poziția corespunzătoare din cadrul fișierului sursă al programului conținutul fișierului de antet specificat între paranteze unghiulare. În cazul nostru, compilatorul va include conținutul fișierului de antet *iostream.h*.

Fișierele cu extensia *.h* pe care le includeți la începutul (sau în *antetul*) unui program sunt *fișiere de antet*. Dacă inspectați directorul care conține fișierele compilatorului, veți găsi un subdirector numit *INCLUDE* care conține un număr mare de fișiere de antet. Fiecare dintre aceste fișiere de antet conține definiții pe care compilatorul le pune la dispoziție pentru diverse operații. Există, de exemplu, un fișier de antet care conține definiții pentru operații matematice, un alt fișier de antet pentru operații cu fișiere și așa mai departe.

Fișierele de antet sunt fișiere ASCII, ceea ce înseamnă că puteți să afișați conținutul acestora pe ecran sau chiar să-l tipăriți. Nu ne va preocupa deocamdată conținutul acestor fișiere de antet. Este suficient să rețineți că instrucțiunea *#include* vă permite folosirea acestor fișiere. Toate programele C++ pe care le veți crea pe parcursul acestei cărți vor conține una sau mai multe instrucțiuni *#include*.

O privire asupra fişierelor de antet C++



Fiecare program C++ pe care îl creaţi va începe cu una sau mai multe instrucţiuni *#include*. Aceste instrucţiuni *#include* determină compilatorul să insereze (includă) în cadrul programului conţinutul unui anume fişier (un fişier de antet), ca şi cum programul ar conţine de fapt instrucţiunile aflate în acel fişier. Fişierele de antet conţin definiţii pe care compilatorul le foloseşte pentru diferite tipuri de operaţii. Există fişiere de antet care definesc operaţii C++ de intrare/ieşire (I/E), servicii ale sistemului de operare (aşa cum sunt funcţiile care întorc data şi ora curente) şi multe altele.

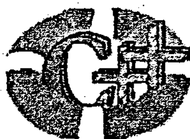
Fişierele de antet sunt, asemeni programelor C++, fişiere ASCII al căror conţinut poate fi afişat pe ecran sau tipărit. Pentru a înţelege mai multe despre conţinutul fişierelor de antet, acordaţi-vă acum câteva minute pentru a tipări fişierul de antet *iostream.h*, al cărui conţinut va fi folosit în fiecare program C++ pe care îl veţi crea pe parcursul acestei cărţi. Fişierul de antet *iostream.h* se află, în mod normal, într-un subdirector numit *INCLUDE* din cadrul directorului ce conţine fişierele compilatorului de C++. Folosiţi un editor de texte pentru a vizualiza şi tipări conţinutul fişierului de antet.

Notă: *Nu alteraţi niciodată conţinutul unui fişier de antet. Acest lucru ar putea duce la generarea de erori de compilator în cadrul programelor pe care le creaţi.*

Despre void main(void)

Atunci când creaţi un program C++, fişierul sursă corespunzător va conţine mai multe instrucţiuni. După cum veţi afla, ordinea în care aceste instrucţiuni apar în program nu este neapărat şi ordinea în care calculatorul va executa instrucţiunile la rularea programului. Dar pentru orice program C++ există un punct de unde începe execuţia programului – *programul principal*. În programele C++, punctul de început al programului este specificat de instrucţiunea *void main(void)*.

Despre programul principal



Fişierele sursă C++ pot conţine multe instrucţiuni. Atunci când rulaţi un program, instrucţiunea *void main(void)* va identifica programul principal, cel care conţine prima instrucţiune pe care o va executa calculatorul. Programele C++ trebuie să aibă întotdeauna o instrucţiune şi numai una care să includă numele *main*.

Atunci când inspectaţi programe C++ mai mari, căutaţi instrucţiunea ce conţine numele *main* pentru a determina instrucţiunile cu care calculatorul va începe execuţia programului.

Pe măsură ce programele dumneavoastră vor deveni din ce în ce mai mari și mai complexe, veți începe să le împărțiți în fragmente mai mici și mai ușor de gestionat. În acel moment, instrucțiunea *void main(void)* va identifica instrucțiunile de bază (sau principale) ale programului – acea parte a programului pe care calculatorul o va executa la început.

Despre semnificația lui void

Pe măsură ce programele vor deveni tot mai complexe, veți începe să împărțiți programele în fragmente mai mici și mai ușor de gestionat, numite *funcții.* O funcție nu este altceva decât o serie de instrucțiuni din program care efectuează o anumită sarcină. De exemplu, dacă creați un program de salarii, ați putea scrie o funcție numită *salariu* care să calculeze retribuția pentru un angajat. Similar, dacă scrieți un program de calcule matematice, ați putea crea funcții numite *radacina_patrata* sau *cub* care să întoarcă rezultatele operațiilor matematice corespunzătoare. Atunci când programul folosește o funcție, aceasta își va îndeplini sarcina și apoi va întoarce rezultatul programului.

Deocamdată, fiecare funcție din program va trebui să aibă un nume unic. De asemenea, fiecare program va avea cel puțin o funcție. Programele pe care le-ați creat în a doua lecție aveau fiecare o singură funcție, *main.* Lecția 10, „O introducere în funcții", se va opri mai îndeaproape asupra funcțiilor. Pentru moment, este suficient să rețineți că o funcție este alcătuită din una sau mai multe instrucțiuni înrudite care îndeplinesc o anumită sarcină.

Pe măsură ce veți inspecta diferite programe C++ veți întâlni cu regularitate cuvântul *void.* Programele folosesc cuvântul *void* pentru a arăta că o funcție nu întoarce nici o valoare sau că programul nu transmite nici un parametru către o funcție.

Majoritatea programelor C++ simple pe care le veți crea pe parcursul acestei cărți nu vor întoarce o valoare de stare la încheiere către sistemul de operare. Din această cauză veți plasa cuvântul *void* înainte de *main,* ca mai jos:

```
void main(void)
```

Programul nu întoarce nici o valoare

În lecțiile următoare veți vedea că programele pot folosi informații (cum ar fi un nume de fișier) pe care utilizatorii le precizează în linia de comandă atunci când rulează respectivele programe. Atunci când un program nu folosește informații din linia de comandă, între parantezele care urmează cuvântul *main* se plasează cuvântul *void,* ca mai jos:

```
void main(void)
```

Programul nu folosește parametri din linia de comandă

Pe măsură ce programele dumneavoastră vor spori în complexitate, ele vor ajunge în cele din urmă să întoarcă valori către sistemul de operare sau să accepte parametri din linia de comandă. Deocamdată, însă, este suficient să precizați înainte de *main* cuvântul cheie *void.*

Utilizarea valorii de stare la încheiere a unui program



În funcție de ceea ce face un program, ar putea fi cazuri în care acesta să fie rulat din cadrul unui fișier de comenzi. După cum execuția programului se încheie sau nu cu succes, ați putea sau nu să doriți ca fișierul de comenzi să ruleze alte programe. Prin intermediul funcției *exit*, programele C++ pot întoarce la încheiere o valoare către sistemul de operare. Un fișier de comenzi poate, apoi, să testeze valoarea de încheiere a programului și să continue corespunzător. Spre exemplu, în MS-DOS, fișierele de comenzi testează valoare de stare la încheiere a unui program cu ajutorul comenzii *IF ERRORLEVEL*. Să presupunem, de pildă, că un program numit *Salarii.EXE* își încheie execuția cu una din următoarele valori de stare la încheiere, în funcție de succesul execuției:

Valoare de stare	Semnificație
0	Succes
1	Fișierul nu a fost găsit
2	Imprimanta nu mai are hârtie

În cadrul unui fișier de comenzi MS-DOS veți putea testa succesul execuției prin intermediul comenzii *IF ERRORLEVEL*, așa cum este ilustrat în continuare:

```
SALARIU
IF ERRORLEVEL 0 IF NOT ERRORLEVEL 1 GOTO SUCCES
IF ERRORLEVEL 1 IF NOT ERRORLEVEL 2 GOTO LIPSA_FISIER
IF ERRORLEVEL 2 IF NOT ERRORLEVEL 3 GOTO LIPSA_HARTIE
REM Aici urmeaza alte comenzi din fisier
```

Despre gruparea instrucțiunilor

Pe măsură ce programele devin din ce în ce mai complexe, ați putea avea o serie de instrucțiuni care doriți să fie executate de către calculator de un anumit număr de ori și ați putea avea o altă serie de instrucțiuni care doriți să fie executate de calculator numai atunci când o condiție anume este îndeplinită. În primul caz, calculatorul ar putea rula aceeași serie de instrucțiuni de 100 de ori pentru a aduna rezultatele la examen a 100 de studenți. În al doilea caz, calculatorul ar putea afișa pe ecran un mesaj atunci când toți studenții au trecut examenul și un alt mesaj atunci când unul sau mai mulți studenți au picat. Pentru a grupa instrucțiuni înrudite din programele C++ veți folosi acoladele deschisă și închisă {}. În cadrul programelor simple prezentate în primele lecții ale cărții, aceste simboluri grupează instrucțiunile care corespund programului principal.

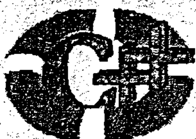
Afişarea pe ecran prin intermediul cout

Toate programele C++ pe care le-aţi creat în cea de a doua lecţie au afişat câte un mesaj pe ecran. Pentru a afişa un mesaj, programele foloseau *cout* şi două simboluri mai-mic consecutive (<<), aşa cum se vede aici:

```
cout << "Hello, C++!";
```

Cuvântul *cout* reprezintă un *flux de ieşire* pe care C++ îl asociază cu dispozitivul de ieşire standard al sistemului de operare. În mod implicit, sistemul de operare asociază dispozitivul standard de ieşire cu ecranul monitorului. Pentru a afişa un mesaj pe ecranul monitorului nu trebuie decât să folosiţi două simboluri mai-mic consecutive (formând ceea ce se numeşte operatorul de inserare) împreună cu fluxul de ieşire *cout*. În lecţia 4, „Afişarea mesajelor pe ecran", veţi vedea că operatorul de inserare poate fi folosit şi pentru a trimite către ecran caractere, numere şi alte simboluri.

Despre fluxul de ieşire cout



După cum aţi văzut, programele C++ folosesc fluxul de ieşire *cout* pentru a afişa mesaje pe ecran. Atunci când afişaţi mesaje prin intermediul lui *cout*, gândiţi-vă la acesta ca la crearea unui flux de caractere pe care sistemul de operare le va afişa pe ecran. Cu alte cuvinte, ordinea în care programul trimite caracterele către *cout* defineşte ordinea în care caracterele vor apărea pe ecran. Ca exemplu, fie următoarele instrucţiuni de program:

```
cout << "Mai intai apare acest mesaj,
cout << "urmat de acest mesaj.
```

În acest caz, programul va afişa fluxul de caractere astfel:

```
Mai intai apare acest mesaj, urmat de acest mesaj.
```

Operatorul de inserare (<<) este numit astfel deoarece acest operator vă permite inserarea de caractere în cadrul fluxului de ieşire.

Aşa cum aţi aflat, fluxul de ieşire *cout* corespunde implicit la ecranul monitorului. Cu alte cuvinte, atunci când programul trimite ceva către *cout*, acel ceva apare pe ecran. Este posibil, însă, ca prin intermediul operatorilor de redirectare ai sistemului de operare să trimiteţi ieşirea unui program către imprimantă sau către un fişier. De exemplu, comanda următoare determină MS-DOS să trimită ieşirea programului *First.EXE* către imprimantă, ca alternativă la ecranul monitorului:

```
C:\> First > PRN <Enter>
```

După cum veţi învăţa în cadrul lecţiei 4, C++ vă permite să utilizaţi *cout* pentru a trimite la ieşire caractere, numere întregi, precum 1001, şi numere în virgulă mobilă, precum 3,12345. În lecţia 7, „Citirea de informaţii de la tastatură", veţi învăţa că C++ oferă şi un flux

de intrare, numit *cin*, pe care programele îl pot folosi pentru a citi informațiile pe care utilizatorul le introduce prin tastatură.

Ce trebuie să știți

Această lecție s-a oprit asupra mai multor elemente uzuale pe care le veți întâlni în programele C++. În lecția 4, „Afișarea mesajelor pe ecran", veți învăța să folosiți *cout* pentru a afișa caractere, numere întregi și numere în virgulă mobilă. Veți afla, de asemenea, și cum pot fi acestea formatate. Dar înainte de a trece la lecția a patra, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Majoritatea programelor C++ încep cu o instrucțiune *#include*, care determină compilatorul să includă în cadrul programului conținutul fișierului de antet specificat.
- ☑ Fișierele de antet conțin definiții pentru compilare pe care le poate utiliza programul dumneavoastră.
- ☑ Un fișier sursă poate conține un număr mare de instrucțiuni. Instrucțiunea *void main(void)* indică începutul programului principal, care conține prima instrucțiune din program care va fi executată.
- ☑ Primul *void* din instrucțiunea *void main(void)* arată compilatorului (și celorlalți programatori care citesc codul) că programul nu întoarce nici o valoare către sistemul de operare.
- ☑ Cel de al doilea *void* din instrucțiunea *void main(void)* arată compilatorului de C++ (ca și celorlalți programatori care citesc codul) că programul nu acceptă parametri din linia de comandă.
- ☑ Pe măsură ce programele devin din ce în ce mai complexe, veți recurge la gruparea instrucțiunilor înrudite sub forma unor fragmente mai mici și mai ușor de gestionat numite funcții.
- ☑ Pentru gruparea instrucțiunilor de program înrudite veți folosi acoladele închisă și deschisă {}.
- ☑ Majoritatea programelor C++ folosesc fluxul de ieșire *cout* pentru a afișa informații pe ecran. Prin intermediul operatorilor de redirectare pentru I/E ai sistemului de operare este posibilă redirectarea ieșirii *cout* către un fișier, către un dispozitiv (așa cum este imprimanta) sau chiar pentru a deveni intrarea unui alt program.

Lecția 4

Afișarea mesajelor pe ecran

Toate programele C++ pe care le-ați creat în lecția a doua, „Crearea primului program", și lecția a treia, „O privire mai atentă asupra limbajului C++", foloseau fluxul de ieșire *cout* pentru afișarea pe ecran a mesajelor (ieșirile programelor). În lecția de față veți utiliza *cout* pentru a afișa caractere, numere întregi, precum 1001, și numere în virgulă mobilă, precum 0,12345. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Programele C++ folosesc fluxul de ieșire *cout* pentru a afișa pe ecran caractere și numere.
- C++ vă permite folosirea unor caractere speciale împreună cu *cout* pentru a genera un tabulator sau o linie nouă, sau chiar pentru a produce un sunet în difuzorul calculatorului.
- C++ permite afișarea într-un mod simplu a numerelor în format zecimal, octal (baza 8) sau hexazecimal (baza 16).
- Prin intermediul operatorilor de redirectare ai sistemului de operare puteți să redirectați mesajele de ieșire pe care programul le trimite către *cout*, de la ecranul monitorului înspre un fișier, o imprimantă sau chiar un al doilea program.
- Prin intermediul fluxului de ieșire *cerr*, programele pot trimite mesaje către dispozitivul de eroare standard, ceea ce evită redirectarea de către utilizatori a mesajelor destinate ecranului.
- Prin intermediul modificatorului *setw* în cadrul unui flux de ieșire, programele au posibilitatea de a formata ieșirile.

Aproape fiecare program C++ pe care îl creați va folosi *cout* pentru afișarea de mesaje pe ecran. Această lecție vă va învăța cum puteți să utilizați la maxim fluxul *cout*.

Utilizarea fluxului cout pentru afișarea numerelor

În cadrul lecțiilor anterioare din această carte, programele foloseau *cout* pentru a afișa pe ecran *șiruri de caractere* (litere și numere încadrate de ghilimele). După cum veți vedea în secțiunea de față, *cout* poate fi folosit și pentru afișarea de numere. Programul următor, *1001.CPP*, afișează pe ecranul monitorului numărul 1001:

```
#include <iostream.h>

void main(void)
   {
     cout << 1001;
   }
```

După ce compilați și executați acest program, pe ecran va fi afișat numărul 1001, așa cum se vede aici:

```
C:\> 1001 <Enter>
1001
```

În continuare, editați programul și modificați instrucțiunea *cout* pentru a afișa numărul 2002, ca mai jos:

```
cout << 2002;
```

Nu uitați că după ce salvați modificările din fișierul sursă trebuie să recompilați programul pentru ca schimbările să fie luate în considerare.

Pe lângă posibilitatea de a afișa *numere întregi* (numere fără virgulă), *cout* permite programelor și afișarea de *numere în virgulă mobilă*, precum 1,2345. Programul care urmează, *Floating.CPP*, folosește *cout* pentru a afișa pe ecran numărul 0,12345:

```
#include <iostream.h>

void main(void)
   {
     cout << 0.12345;
   }
```

Ca mai devreme, compilați și executați programul *Floating.CPP*. În replică, pe ecran vor fi afișate următoarele:

```
C:\> Floating <Enter>
0.12345
```

Afișarea simultană a mai multor valori

Programatorii de C++ se referă la cele două simboluri mai-mic consecutive ca la *operatorul de inserare* (acest operator inserează caractere în fluxul de ieșire pentru a fi afișate). Atunci când folosiți *cout* puteți specifica operatorul de inserare de mai multe ori într-o singură instrucțiune. De exemplu, programul următor, *1001Too.CPP*, folosește operatorul de inserare de patru ori pentru a afișa pe ecran numărul 1001:

```
#include <iostream.h>

void main(void)
   {
      cout << 1 << 0 << 0 << 1;
   }
```

După compilarea şi rularea programului *1001Too.CPP*, pe ecran vor fi afişate următoarele:

```
C:\> 1001Too <Enter>
1001
```

De fiecare dată când C++ întâlneşte operatorul de inserare, numărul sau caracterele care urmează sunt pur şi simplu ataşate celor aflate curent în fluxul de ieşire. Programul următor, *Show1001.CPP*, afişează un şir de caractere şi un număr prin intermediul lui *cout*:

```
#include <iostream.h>

void main(void)
   {
   cout << "My favorite number is " << 1001;
   }
```

Remarcaţi cum caracterul spaţiu ce urmează cuvântului *is* (în cadrul ghilimelelor) este folosit pentru a plasa numărul 1001 cu un spaţiu după acest cuvânt. În lipsa spaţiului, numărul ar apărea imediat după cuvânt (is1001). În mod similar, următorul program, *1001Mid.CPP*, afişează numărul 1001 în mijlocul unui şir de caractere:

```
#include <iostream.h>

void main(void)
   {
      cout << "The number  << 1001 <<  is my favorite";
   }
```

Ca şi mai înainte, observaţi spaţiile din şirurile de caractere care apar înainte şi după 1001. Dacă nu aţi include caracterele spaţiu de după cuvântul *number* şi dinaintea cuvântului *is*, programul ar plasa numărul 1001 lipit de aceste cuvinte (number1001is).

În fine, programul următor, *MixMatch.CPP*, combină şiruri, caractere, numere întregi şi numere în virgulă mobilă în cadrul aceluiaşi flux de ieşire:

```
#include <iostream.h>

void main(void)
  {
    cout << "At age " << 20 << " my salary was "
      << 493.34 << endl;
  }
```

După ce compilați și rulați programul *MixMatch.CPP*, pe ecran vor afișate următoarele:

```
C:\> MixMatch <Enter>
At age 20 my salary was 493.34
```

Utilizarea caracterelor de ieșire speciale

Toate programele pe care le-ați creat până acum în această carte afișau mesaje pe o singură linie. Majoritatea programelor pe care le veți crea în viitor vor afișa, însă, mai multe linii de ieșire. De exemplu, să presupunem că scrieți un program care afișează pe ecran adresa dumneavoastră. Veți dori probabil ca adresa să apară pe mai multe rânduri – exact așa cum ar apărea pe un plic.

Atunci când vreți să plasați cursorul la începutul liniei următoare puteți să plasați în fluxul de ieșire *caracterul de linie nouă* (\n). C++ vă oferă două modalități de generare a caracterului de linie nouă. O posibilitate este plasarea caracterului \n în interiorul unui șir de caractere. De exemplu, programul următor, *TwoLines.CPP*, inserează caracterul de linie nouă în șirul de ieșire, ceea ce face ca mesajul programului să fie afișat pe două linii:

```
#include <iostream.h>

void main(void)
  {
    cout < "This is line one\nThis is line two";
  }
```

Atunci când compilați și executați programul *TwoLines.CPP*, caracterul de linie nouă va face ca programul să afișeze la ieșire două linii, ca mai jos:

```
C:\> TwoLines <Enter>
This is line one
This is line two
```

Dacă programul dumneavoastră nu afișează un șir de caractere, atunci puteți plasa caracterul de linie nouă între două apostrofuri. Spre exemplu, următorul program, *NewLines.CPP*, afișează numerele 1, 0, 0 și 1 fiecare pe câte o linie:

```
#include <iostream.h>

void main(void)
  {
    cout << 1 << '\n' << 0 << '\n' << 0 << '\n' << 1;
  }
```

C++ tratează simbolul de linie nouă ca pe un caracter, asemeni literelor 'a', 'b' sau 'c'. Din această cauză, atunci când folosiți caracterul de linie nouă în afara unui șir de caractere, trebuie să încadrați acest caracter între apostrofuri. Pe lângă utilizarea caracterului de linie nouă pentru avansarea cursorului la începutul liniei următoare, programele pot folosi și simbolul *endl* (end line – sfârșit de linie). Programul care urmează, *Endl.CPP*, ilustrează modul în care poate fi utilizat *endl* pentru a plasa cursorul la începutul unei noi linii:

```
#include <iostream.h>

void main(void)
  {
    cout << "I've been..." << endl << "Rescued by C++";
  }
```

Ca și mai devreme, după compilarea și rularea programului *Endl.CPP*, pe ecran va fi afișată ieșirea programului pe două linii:

```
C:\> Endl <Enter>
I've been
Rescued by C++
```

În fine, programul următor, *Address.CPP*, afișează pe mai multe linii adresa Jamsa Press:

```
#include <iostream.h>

void main(void)
  {
    cout << "Jamsa Press" << endl;
    cout << "2975 South Rainbow, Suite I" << endl;
    cout << "Las Vegas, NV 89102" << endl;
  }
```

În acest caz, programul *Address.CPP* folosește trei instrucțiuni pentru afișarea adresei. Dar după cum se poate vedea, programul ar putea folosi o singură instrucțiune care să afișeze pe ecran întreg mesajul, ca mai jos:

```
void main(void)
   {
      cout << "Jamsa Press" << endl << "2975
        South Rainbow, Suite I" << endl << "Las Vegas,
        NV 89102" << endl;
   }
```

Precum vedeți, programul folosește o singură instrucțiune pentru a plasa adresa în cadrul fluxului de ieșire *cout*. Deși cele două programe îndeplinesc aceeași operație, majoritatea programatorilor vor considera că primul program, cel care folosește trei instrucțiuni, este mai ușor de înțeles.

Alte caractere speciale

Pe lângă caracterul de linie nouă, care permite programelor să avanseze cursorul la începutul unei linii noi, în cadrul fluxului de ieșire *cout* pot fi folosite și caracterele speciale enumerate în tabelul 4.

Caracter	Scop
\a	Caracter de avertizare (sau sunet)
\b	Caracter backspace
\f	Caracter de avans la pagină
\n	Caracter de linie nouă
\r	Caracter de retur de car (fără avans de linie)
\t	Caracter de tabulator orizontal
\v	Caracter de tabulator vertical
\\	Caracter backslash
\?	Caracter semn de întrebare
\'	Caracter apostrof
\"	Caracter ghilimele
\0	Caracter nul
\ooo	Valoare octală, cum ar fi *\007*
\xhhhh	Valoare hexazecimală, cum ar fi *\xFFFF*

Tabelul 4 *Caracterele speciale pe care programele le pot folosi în cadrul fluxului de ieșire* ***cout***.

Notă: *La utilizarea caracterelor speciale enumerate în tabelul 4, aceste caractere trebuie încadrate de apostrofuri atunci când le folosiți independent, cum ar fi '\n', respectiv de ghilimele atunci când le folosiți în interiorul unui șir, cum ar fi „Hello\nWorld!"*

Programul următor, *Special.CPP*, folosește caracterele speciale de avertizare (\a) și de tabulare (\t) pentru a genera sunete în difuzorul calculatorului și afișa apoi cuvintele *Bell Bell Bell*, separate de câte un tabulator:

```
#include <iostream.h>

void main(void)
  {
    cout << "Bell\a\tBell\a\tBell\a";
  }
```

Afișarea valorilor octale și hexazecimale

Până acum, programele din această lecție au afișat numere zecimale. În funcție de scopul unui program, ar putea fi cazuri în care să fie necesară afișarea unor valori octale sau hexazecimale. În acest scop, programele dumneavoastră pot recurge la inserarea în cadrul fluxului de ieșire a modificatorilor *dec*, *oct* și *hex*. Programul următor, *OctHex.CPP*, folosește acești trei modificatori pentru a afișa valorile zecimale 10 și 20 în octal și în hexazecimal:

```
#include <iostream.h>

void main(void)
  {
    cout << "Octal: " << oct << 10 << ' ' << 20 << endl;
    cout << "Hexadecimal: " << hex << 10 << ' ' << 20
      << endl;
    cout << "Decimal: " << dec << 10 << ' ' << 20 << endl;
  }
```

După compilarea și executarea programului *OctHex.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> OctHex <Enter>
Octal: 12 24
Hexadecimal: a 14
Decimal: 10 20
```

Notă: *Atunci când utilizați unul dintre modificatorii de ieșire pentru a selecta formatul octal, hexazecimal sau zecimal, selecția făcută va rămâne valabilă până la încheierea programului sau până la utilizarea unui alt modificator.*

Scrierea către dispozitivul de eroare standard

Așa cum ați învățat, dacă un program folosește *cout* pentru afișarea rezultatelor, ieșirea respectivului program poate fi apoi redirectată către un dispozitiv sau către un fișier prin intermediul operatorilor de redirectare ai sistemului de operare. Atunci când programul detectează o eroare, însă, nu veți vrea ca sistemul de operare să redirecteze mesajele de eroare îndreptate către ecran. De exemplu, în cazul în care sistemul de operare redirecteză mesajele de eroare către un fișier, este posibil ca utilizatorul să nu fie conștient despre apariția unei erori.

Dacă vreți ca un program să afișeze pe ecran un mesaj de eroare, atunci ar trebui să apelați la fluxul de ieșire *cerr*. C++ asociază *cerr* cu dispozitivul de eroare standard al sistemului de operare. Programul următor, *Cerr.CPP*, folosește fluxul de ieșire *cerr* pentru a afișa pe ecran mesajul „*This Message Always Appears*":

```
#include <iostream.h>

void main(void)
  {
    cerr << "This Message Always Appears";
  }
```

Compilați și rulați programul *Cerr.CPP*. Apoi încercați redirectarea ieșirii programului către un fișier prin intermediul operatorului de redirectare, așa cum se ilustrează în continuare:

```
C:\> Cerr > FileName.EXT <Enter>
```

Deoarece sistemul de operare nu va permite programului să redirecteze ieșirea trimisă către dispozitivul de eroare standard, mesajul va apărea pe ecranul monitorului.

Fixarea lățimii de afișare

Mai multe din programele anterioare au folosit fluxul de ieșire *cout* în scopul afișării pe ecran a unor numere. Pentru a asigura afișarea corectă a numerelor (cu o spațiere corespunzătoare), programele au inserat caractere spațiu înainte și după numere. Atunci când folosiți pentru afișare *cout* sau *cerr*, programele pot specifica lățimea de afișare pentru fiecare valoare prin intermediul modificatorului *setw* (set width – fixarea lățimii).

Prin *setw*, programele specifică numărul minim de caractere care vor fi folosite pentru afișarea valorii. De exemplu, programul următor, *Setw.CPP*, folosește modificatorul *setw* pentru a selecta lățimi de 3, 4, 5 și 6 caractere pentru valoarea 1001. Pentru a putea utiliza modificatorul *setw* va trebui să includeți la începutul programului fișierul de antet *iomanip.h*, așa cum se vede în continuare:

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
  {
  cout << "My favorite number is" << setw(3) << 1001 << endl;
  cout << "My favorite number is" << setw(4) << 1001 << endl;
  cout << "My favorite number is" << setw(5) << 1001 << endl;
  cout << "My favorite number is" << setw(6) << 1001 << endl;
  }
```

După ce compilaţi şi rulaţi programul *Setw.CPP*, pe ecran vor fi afişate următoarele:

```
C:\> Setw <Enter>
My favorite number is1001
My favorite number is1001
My favorite number is 1001
My favorite number is  1001
```

Atunci când specificaţi o lăţime prin intermediul lui *setw*, ceea ce precizaţi este numărul *minim* de caractere cu care va fi afişată o valoare. În cazul programului *Setw.CPP*, modificatorul *setw(3)* fixează un minim de trei caractere. Cum valoarea 1001 necesită mai mult de trei caractere, *cout* a folosit numărul de caractere necesare în fapt, adică patru. De asemenea, atunci când recurgeţi la *setw* pentru fixarea unei lăţimi, fluxul de ieşire va folosi respectiva lăţime numai pentru afişarea valorii care urmează. Dacă trebuie fixate lăţimi pentru mai multe valori, va trebui să folosiţi *setw* de mai multe ori.

Notă: Programul ***Setw.CPP*** *foloseşte fişierul de antet* ***iomanip.h***. *Ar putea fi o idee bună să tipăriţi şi să examinaţi conţinutul acestui fişier. Ca şi în cazul fişierului de antet* ***iostream.h***, *fişierul în cauză ar trebui să se afle în subdirectorul* ***INCLUDE*** *din cadrul directorului care conţine fişierele compilatorului.*

Ce trebuie să ştiţi

În această lecţie aţi învăţat să folosiţi *cout* sub mai multe forme în scopul afişării pe ecran. Fiecare dintre programele pe care le veţi crea în continuare pe parcursul acestei cărţi va folosi pentru afişare *cout*. În lecţia 5, „Programele păstrează informaţii în variabile", veţi învăţa să folosiţi variabile în cadrul programelor pentru a păstra valori ce se pot modifica pe parcursul execuţiei. Dar înainte de a continua cu lecţia 5, asiguraţi-vă că aţi reţinut următoarele aspecte importante:

☑ Fluxul de ieşire *cout* vă permite afişarea de caractere şi numere către dispozitivul de ieşire standard al sistemului de operare – care este, în mod normal, ecranul.

- ☑ Prin utilizarea unor simboluri speciale în cadrul fluxului de ieșire, programele pot specifica o linie nouă, un tabulator sau alte caractere deosebite.
- ☑ Pentru a avansa cursorul la începutul liniei următoare, programele pot crea o linie nouă prin intermediul caracterului de linie nouă \n sau al modificatorului *endl.*
- ☑ Modificatorii *dec*, *oct* și *hex* fac posibilă afișarea de către programe a valorilor sub formă zecimală, octală și hexazecimală.
- ☑ Prin intermediul fluxului de ieșire *cerr*, programele pot scrie mesaje către dispozitivul de eroare standard al sistemului de operare – ieșirea acestuia nu poate fi redirectată de către utilizator.
- ☑ Prin intermediul modificatorului *setw* este posibilă fixarea în programe a lățimii de afișare pentru valori.

Lecția 5

Programele păstrează informații în variabile

Toate programele prezentate începând cu lecția a doua, „Crearea primului program", și terminând cu lecția a patra, „Afișarea mesajelor pe ecran", au fost foarte simple. Pe măsură ce încep să facă activități utile, programele sunt nevoite ca pe parcursul rulării să rețină anumite informații. De exemplu, un program care tipărește un fișier trebuie să știe numele fișierului și, eventual, numărul de copii de tipărit. Atunci când rulează, programul păstrează astfel de informații în memoria cu acces aleatoriu (RAM) a calculatorului. Pentru a stoca și extrage informații din locații de memorie specifice, programele folosesc *variabile.* Sub forma cea mai simplă, o variabilă reprezintă numele unei locații de memorie care poate păstra o valoare anume. Lecția de față studiază modul de creare și de utilizare a variabilelor în cadrul programelor C++. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Variabilele pe care le folosiți într-un program trebuie declarate prin informarea compilatorului despre numele și tipul acestora.
- *Tipul* unei variabile indică genul de valoare (cum ar fi un număr întreg sau în virgulă mobilă) pe care variabila o poate păstra și operațiile pe care programul le poate efectua cu acea variabilă.
- Pentru a atribui o valoare unei variabile veți folosi operatorul de atribuire din C++ (semnul egal).
- Pentru a afișa pe ecran valoarea unei variabile, programele folosesc fluxul de ieșire *cout.*
- La declararea variabilelor este recomandat să folosiți nume sugestive pentru a vă face programele mai ușor de citit și de înțeles de către alți programatori.
- În interiorul programului este bine să plasați comentarii care descriu și pentru alți programatori operațiile efectuate. Astfel, dacă un alt programator este nevoit să modifice programul dumneavoastră, comentariile pe care le-ați inserat vor descrie în detaliu operațiile efectuate de program.

Gândiți-vă la o variabilă ca la o cutie în care puteți plasa o valoare. Atunci când atribuiți o valoare variabilei, plasați o valoare în interiorul cutiei. Atunci când folosiți ulterior valoarea variabilei, calculatorul va apela pur și simplu la valoarea conținută în cutie.

Declararea variabilelor în cadrul programului

Pentru păstrarea informațiilor, programele folosesc variabile. În funcție de genul de valoare pe care doriți să o păstrați, cum ar fi un număr întreg, o literă a alfabetului sau o valoare în virgulă mobilă, *tipul* variabilei poate varia. Tipul unei variabile indică genul de

valoare pe care o poate păstra variabila, precum și mulțimea operațiilor (ca adunarea, înmulțirea și altele) pe care programul le poate efectua cu valoarea acelei variabile. Cele mai multe programe C++ folosesc tipurile de variabile enumerate în tabelul 5.1.

Tip	Valori păstrate
char	Valori aflate între –128 și 127. Programele folosesc de regulă tipul *char* pentru a stoca literele alfabetului.
int	Valori aflate între –32.768 și 32.767.
unsigned	Valori aflate între 0 și 65.535.
long	Valori aflate între –2.147.483.648 și 2.147.483.647.
float	Valori aflate între $-3{,}4 \times 10^{-38}$ și $3{,}4 \times 10^{38}$.
double	Valori aflate între $-1{,}7 \times 10^{-308}$ și $1{,}7 \times 10^{308}$.

Tabelul 5.1 *Tipurile uzuale de variabile din C++.*

Înainte ca programul să poată folosi o variabilă, acea variabilă trebuie *declarată*. Cu alte cuvinte, programul trebuie să prezinte variabila compilatorului de C++. Pentru declararea unei variabile într-un program trebuie să specificați tipul variabilei și numele pe care programul îl va folosi pentru a referi respectiva variabilă. De exemplu, după acolada care deschide programul principal, puteți specifica tipul și numele variabilei, ca mai jos:

```
tipul_variabilei numele_variabilei;
```

De obicei, tipul variabilei va fi unul dintre tipurile prezentate în tabelul 5.1. Numele variabilei este un nume sugestiv pe care îl alegeți dumneavoastră pentru a descrie (cuiva care citește programul) scopul acelei variabile. Spre exemplu, programele ar putea folosi variabile precum *nume_angajat*, *varsta_angajat* și așa mai departe. Observați semnul punct și virgulă care urmează numelui de variabilă din declarația variabilei. C++ consideră că declararea unei variabile este o instrucțiune. Din acest motiv, la sfârșitul declarației trebuie plasat semnul punct și virgulă.

Fragmentul de program următor declară trei variabile având tipurile *int*, *float* și *long*:

```
#include <iostream.h>

void main(void)
  {
  int test_score;
  float salary;
  long distance_to_mars;
  }
```

Este important să remarcaţi că acest program nu face altceva decât să declare cele trei variabile. Cu alte cuvinte, dacă veţi compila şi rula acest fragment de cod, nu va fi afişat nimic. După cum puteţi vedea, fiecare declaraţie de variabilă se încheie cu punct şi virgulă.

Atunci când declaraţi mai multe variabile de acelaşi tip, C++ vă permite enumerarea numelor de variabile, separate prin virgule. Instrucţiunea următoare, de pildă, declară trei variabile de tip virgulă mobilă:

```
float salary, income_tax, retirement_fund;
```

Despre variabile



O variabilă este numele unei locaţii de stocare din memoria cu acces aleatoriu (RAM) a calculatorului. Atunci când rulează, un program păstrează informaţiile în variabile. La crearea programelor trebuie să declaraţi variabilele prin indicarea compilatorului de C++ a tipului şi numelor variabilelor. Instrucţiunea următoare, spre exemplu, declară o variabilă de tip *int* al cărei nume este *age*:

```
int age;
```

Aşa cum veţi vedea, tipul unei variabile indică mulţimea de valori pe care variabila le poate stoca (de pildă, numere întregi sau în virgulă mobilă) şi mulţimea de operaţii pe care programul le poate efectua cu acea variabilă (precum adunarea sau scăderea).

Folosiţi nume de variabile sugestive

Fiecare variabilă pe care o creaţi într-un program trebuie să aibă un nume unic. Pentru ca programele dumneavoastră să fie mai uşoare de citit şi de înţeles, ar trebui să alegeţi nume de variabile sugestive. De exemplu, instrucţiunea următoare declară trei variabile numite *x*, *y* şi *z*:

```
int x, y, z;
```

Presupunând că variabilele ar păstra vârsta, nota şi clasa unui elev, numele de variabile următoare sunt mai mult semnificative pentru un alt programator care vă citeşte codul sursă:

```
int varsta_elev, nota, clasa;
```

Atunci când alegeţi numele de variabile, puteţi folosi combinaţii de litere, cifre şi linii de subliniere (_). Primul caracter din numele variabilei trebuie să fie o literă sau o linie de subliniere. Nu puteţi să începeţi numele unei variabile cu o cifră. De asemenea, C++

tratează literele minuscule și cele majuscule ca fiind diferite. Pentru început folosiți în numele de variabile numai litere mici. Pe măsură ce vă obișnuiți cu C++, puteți începe să combinați literele mici și cele mari pentru a crea nume sugestive, ca mai jos:

```
float SalariuLunar, TaxaPeVenit;
```

Cuvinte ce nu pot reprezenta nume de variabile

Atunci când creați nume de variabile, trebuie să știți că C++ rezervă cuvintele enumerate în tabelul 5.2 ca și cuvinte cheie ce au o semnificație specială pentru compilator. Nu aveți voie să folosiți un cuvânt cheie din C++ ca nume de variabilă.

Cuvinte cheie din C++

asm	*auto*	*break*	*case*	*catch*	*char*
class	*const*	*continue*	*default*	*delete*	*do*
double	*else*	*enum*	*extern*	*float*	*for*
friend	*goto*	*if*	*inline*	*int*	*long*
new	*operator*	*private*	*protected*	*public*	*register*
return	*short*	*signed*	*sizeof*	*static*	*struct*
switch	*template*	*this*	*throw*	*try*	*typedef*
union	*unsigned*	*virtual*	*void*	*volatile*	*while*

Tabelul 5.2 *Cuvinte cheie din C++.*

De ce folosesc programele variabile



Pe măsură ce programele dumneavoastră vor deveni din ce în ce mai complexe, operațiile efectuate ar putea implica mai multe elemente. De exemplu, în cazul în care scrieți un program de salarii, acesta ar trebui să prelucreze informațiile pentru fiecare angajat. Într-un astfel de program ați putea folosi variabile pe care să le denumiți *nume_angajat*, *id_angajat*, *salariu_angajat* și așa mai departe. La lansarea sa, programul va atribui acestor variabile informațiile despre primul angajat. După calcularea salariului acestui angajat, programul va repeta procesul pentru angajatul următor. Pentru a calcula salariul celui de al doilea angajat, programul va atribui informațiile corespunzătoare acestui angajat (numele său, identificatorul și salariul) variabilelor enumerate mai devreme, după care va efectua prelucrările de rigoare. Cu alte cuvinte, pe parcursul execuției sale, programul atribuie variabilelor valori diferite – ceea ce va duce la schimbarea sau „variația" valorilor din variabile.

Atribuirea unei valori pentru o variabilă

Aşa cum aţi aflat, variabilele păstrează valori pe parcursul execuţiei programului. După declararea unei variabile, pentru atribuirea unei valori acelei variabile veţi folosi *operatorul de atribuire* (semnul egal). Instrucţiunile următoare atribuie valori mai multor variabile diferite. Remarcaţi prezenţa semnului punct şi virgulă la sfârşitul fiecărei instrucţiuni, aşa cum se vede aici:

```
age = 32;
salary = 25000.75;
distance_to_the_moon = 238857;
```

Notă: *La atribuirea de valori variabilelor nu folosiţi virgule în cadrul valorilor (cum ar fi 25,000.75 sau 238,857). Dacă includeţi virgule, compilatorul de C++ va genera şi va afişa mesaje de eroare.*

Fragmentul de program care urmează declară variabilele descrise mai devreme şi foloseşte apoi operatorul de atribuire pentru a atribui valori acestor variabile.

```
#include <iostream.h>

void main(void)
  {
  int age;
  float salary;
  long distance_to_the_moon;

  age = 32;
  salary = 25000.75;
  distance to the moon = 238857;
  }
```

Încă o dată, este important să observaţi că acest program nu afişează nimic. În schimb, scopul programului este să vă arate cum puteţi atribui valori pentru una sau mai multe variabile.

Atribuirea unei valori la declarare

Atunci când declaraţi o variabilă, este de multe ori convenabil să atribuiţi în acelaşi timp şi valoarea iniţială a variabilei (programatorii numesc acest proces „iniţializarea variabilei"). Pentru a înlesni iniţializarea variabilelor, C++ vă permite atribuirea de valori odată cu declararea variabilelor, aşa cum se vede aici:

```
int age = 32;
float salary = 25000.75;
long distance_to_the_moon = 238857;
```

Multe dintre programele prezentate în această carte vor atribui valori variabilelor odată cu declararea acestora.

Atribuirea de valori unei variabile



Variabilele păstrează informații pe durata execuției unui program. Pentru a plasa o valoare în cadrul unei variabile, programele trebuie să apeleze la operatorul de atribuire (semnul egal) din C++. Instrucțiunea următoare folosește operatorul de atribuire pentru a atribui valoarea 5 variabilei *lectie*:

```
lectie = 5;
```

Pentru a simplifica procesul de atribuire de valori variabilelor, C++ vă permite totodată atribuirea unei valori pentru o variabilă odată cu declararea acelei variabile, așa cum este ilustrat în continuare:

```
int lectie = 5;
```

Utilizarea valorii unei variabile

Odată ce ați atribuit o valoare unei variabile, programele pot folosi valoarea acelei variabile prin simpla specificare a numelui de variabilă. Programul următor, *ShowVars.CPP*, atribuie valori pentru trei variabile și apoi afișează valoarea fiecărei variabile prin *cout*:

```
#include <iostream.h>

void main(void)
  {
    int age = 32;
    float salary = 25000.75;
    long distance to the moon = 238857;
    cout << "The employee is " << age << " years old"
      << endl;
    cout << "The employee makes $" << salary << endl;
    cout << "The moon is " << distance to the moon <<
      " miles from the earth" << endl;
  }
```

Notă: *Ultima instrucțiune* ***cout*** *nu încape pe o singură linie. În acest caz, programul desfășoară pur și simplu cuvintele pe cea de a doua linie. Deoarece C++ folosește semnul punct și virgulă pentru a marca sfârșitul unei instrucțiuni, o astfel de desfășurare pe linii este posibilă. Atunci când trebuie să desfășurați conținutul unei linii pe linia următoare, aveți grijă să nu întrerupeți linia în interiorul unui șir de caractere (între ghilimele), iar după aceea indentați noua linie cu unul sau două spații, așa cum se vede.*

La compilarea și rularea programului *ShowVars.CPP*, ecranul va afișa următoarele:

```
C:\> ShowVars <Enter>
The employee is 32 years old
The employee makes $25000.75
The moon is 238857 miles from the earth
```

Precum vedeți, pentru utilizarea valorii unei variabile este suficientă referirea numelui acelei variabile în cadrul programului. Înainte de a trece mai departe, acordați-vă câteva minute pentru a modifica codul sursă al programului prin atribuirea de diferite valori variabilelor *age* și *salary*, ca mai jos:

```
int age = 44;
float salary = 52000.50;
```

Compilați și rulați programul. Prin modificarea valorilor variabilelor se schimbă și ieșirea programului, după cum se vede aici:

```
C:\> ShowVars <Enter>
The employee is 44 years old
The employee makes $52000.50
The moon is 238857 miles from the earth
```

Depășirea capacității de stocare a unei variabile

După cum ați aflat, tipul unei variabile definește mulțimea de valori pe care acea variabilă le poate reține. De exemplu, o variabilă de tip *int* poate reține valori aflate între –32.768 și 32.767. Atunci când atribuiți unei variabile o valoare ce se află în afara intervalului corespunzător tipului variabilei, se va produce o eroare de *depășire*. Spre exemplu, programul următor, *Overflow.CPP*, ilustrează modul în care depășirea intervalului de valori admise pentru o variabilă duce la o eroare. Programul va atribui valoarea 40.000 unei variabile de tip *int*, pentru care valoarea maximă permisă este 32.767. Apoi, programul va atribui valoarea 40.000.000.000 unei variabile de tip *long*, pentru care valoarea maximă permisă este 2.147.483.467. În fine, programul va atribui valoarea 210 unei variabile de tip *char*, pentru care valoarea maximă permisă este 127, toate acestea fiind prezentate în continuare:

```
#include <iostream.h>

void main(void)
  {
    int positive = 40000;
    long big positive = 4000000000;
    char little_positive = 210;

    cout << "positive now contains " << positive << endl;
    cout << "big positive now contains " << big positive
      << endl;
    cout << "little positive now contains "
      << little positive << endl;
  }
```

Atunci când compilați și rulați programul *Overflow.CPP*, pe ecran vor fi afișate următoarele:

```
positive now contains -25536
big_positive now contains -294967296
little_positive now contains Ò
```

Așa cum puteți vedea, programul atribuie variabilelor de tip *int*, *long* și *char* valori care depășesc intervalul admis – ceea ce duce la o eroare de depășire. Atunci când lucrați cu variabile trebuie să țineți cont de intervalul de valori pe care tipul unei variabile le permite. Erorile de depășire sunt subtile și pot fi dificil de detectat și corectat. De asemenea, remarcați valoarea pe care programul o afișează pentru variabila *little_positive*. Pentru că tipul acestei variabile este *char*, fluxul de ieșire *cout* încearcă afișarea valorii variabilei sub forma unui caracter. În acest caz, valoarea afișată corespunde caracterului ASCII extins ce are codul 210.

Despre precizie

În secțiunea anterioară ați văzut că erorile de depășire apar atunci când atribuiți unei variabile o valoare care se află în afara intervalului de valori pe care tipul acelei variabile le permite. În mod similar, trebuie să știți că *precizia* (acuratețea) cu care calculatoarele pot stoca numere nu este nelimitată. De exemplu, când lucrați cu numere în virgulă mobilă (valori cu virgulă), există cazuri în care calculatorul nu poate reprezenta numărul în forma sa exactă. Asemenea erori de precizie pot fi dificil de sesizat în cadrul programelor.

Programul următor, *Precise.CPP*, atribuie o valoare mai mică de 0,5 unor variabile de tip *float* și *double*. Din nefericire, cum capacitatea calculatorului de a reprezenta numere este limitată, variabilele din program nu conțin de fapt valoarea atribuită prin program, ci valoarea 0,5, așa cum se vede aici:

```
#include <iostream.h>

void main(void)
  {
    float f_not_half = 0.49999990;
    double d_not_half = 0.49999990;

    cout << "Floating point 0.49999990 is "
      << f_not_half << endl;
    cout << "Double 0.49999990 is " << d_not_half
      << endl;
  }
```

După compilarea și rularea programului *Precise.CPP*, pe ecran vor fi afișate următoarele:

```
Floating point 0.49999990 is 0.5
Double 0.49999990 is 0.5
```

După cum vedeți, valorile pe care programul le atribuie variabilelor și valorile pe care variabilele le conțin de fapt nu sunt exact aceleași. Asemenea erori de precizie apar deoarece calculatorul trebuie să reprezinte numerele prin intermediul unui număr fix de unu și zero. În multe cazuri, calculatorul reușește să reprezinte numerele exact. În alte situații, așa cum o arată și programul, reprezentarea din calculator este apropiată, dar nu exactă. Atunci când programați trebuie să țineți cont și de precizie. În funcție de valorile cu care lucrează programele, este posibil să apară erori de precizie care sunt foarte dificil de detectat.

Folosiți comentarii pentru a spori lizibilitatea programelor

Atunci când programele devin mai complexe, numărul de instrucțiuni pe care acestea le conțin pot face respectivele programe dificil de înțeles. Deoarece alți programatori ar putea fi nevoiți să înțeleagă și chiar să modifice programele dumneavoastră, va trebui să scrieți programele sub cea mai lizibilă formă posibilă. Lizibilitatea unui program poate fi sporită prin următoarele căi:

- Folosiți nume de variabile sugestive pentru a descrie scopul variabilelor
- Păstrați o indentare și o aliniere corespunzătoare a instrucțiunilor (vedeți lecția 8)
- Folosiți linii goale pentru a separa instrucțiunile care nu au legătură între ele
- Folosiți comentarii care explică operațiile efectuate de program

Odată cu crearea unui program, aveți posibilitatea de a plasa în cadrul fișierului sursă remarci ce explică operațiile efectuate de program. Asemenea remarci (programatorii le numesc *comentarii*) nu numai că ajută alți programatori să vă înțeleagă programul, dar v-ar putea ajuta și pe dumneavoastră să vă amintiți, la câteva luni de zile de la ultima inspectare a programului, de ce ați folosit anumite instrucțiuni. Pentru a plasa un

comentariu într-un program C++ nu trebuie decât să introduceți două semne slash (//) printre instrucțiunile programului, ca mai jos:

```
// Acesta este un comentariu
```

Atunci când întâlnește cele două semne slash, compilatorul ignoră tot textul aflat în continuare pe acea linie. Este recomandat să plasați, cel puțin, comentarii la începutul fiecărui program pentru a arăta cine a scris programul, când și de ce, așa cum se ilustrează mai jos:

```
// Program: Buget,CPP
// Programator: Kris Jamsa
// Data crearii: 10.01.98
//
// Scop: Urmareste informatiile lunare despre buget.
```

Pe măsură ce un program efectuează diferite operații, ar trebui să plasați comentarii înainte sau după anumite instrucțiuni pentru a explica scopul acestora. Să luăm, de exemplu, următoarea instrucțiune de atribuire:

```
distance_to_the_moon = 238857;          // Distanta in mile
```

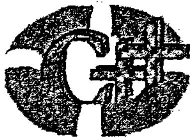
Comentariul din dreapta instrucțiunii de atribuire oferă informații suplimentare celui care va parcurge programul. În cazul instrucțiunii de mai sus, comentariul informează un alt programator care citește codul că valoarea 238.857 reprezintă distanța în mile până la Lună.

Programatorii începători au deseori dificultăți în a determina când și ce anume să comenteze. Ca o regulă, *nu trebuie să aveți prea multe comentarii în programe.* Asigurați-vă, în schimb, că acestea sunt sugestive. Comentariile care urmează nu oferă nici un fel de informații suplimentare unui programator care vă citește codul:

```
age = 32;                    // Atribuim 32 variabilei age
salary = 25000.75;           // Atribuim 25000.75
                             // variabilei salary
```

Scopul în care veți folosi comentarii este să explicați de ce anume sunt efectuate anumite operații.

Adăugarea de comentarii în programe



Odată cu crearea programelor, este bine să inserați comentarii care să explice operațiile efectuate de către acestea. Dacă alți programatori ar fi nevoiți să modifice programul dumneavoastră, aceștia pot apela la comentariile dumneavoastră pentru a înțelege comportamentul programului. Programele C++ folosesc, de regulă, două semne slash pentru a indica un comentariu, ca mai jos:

```
// Acesta este un comentariu in C++
```

Atunci când întâlnește două semne slash, compilatorul de C++ va ignora orice text (rămas pe linia curentă) care urmează slash-urilor. Programele bune ar trebui să fie ușor de citit și de înțeles. Comentariile sporesc lizibilitatea unui program.

Notă: *Pe lângă utilizarea comentariilor pentru creșterea lizibilității unui program, este bine să folosiți și linii goale pentru a separa instrucțiunile de program care nu au legătură între ele. La întâlnirea unei linii goale, compilatorul de C++ trece pur și simplu la linia următoare.*

Ce trebuie să știți

În această lecție ați învățat că programele păstrează pe durata rulării informațiile în cadrul variabilelor. Pe scurt, o variabilă este un nume pe care programul îl asociază cu o locație de memorie în care va depune informația. Înainte ca un program să poată folosi o variabilă, trebuie să declarați numele și tipul variabilei. În lecția 6, „Efectuarea de operații elementare", veți învăța să efectuați operații simple cu variabile, precum adunarea și scăderea. Dar înainte de a trece lecția 6, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Pentru a utiliza variabile în cadrul unui program trebuie să declarați tipul și numele acestora.
- ☑ Numele de variabile trebuie să fie unice și sugestive pentru un alt programator care citește codul sursă. Numele unei variabile ar trebui să corespundă scopului acesteia.
- ☑ Numele de variabile trebuie să înceapă cu o literă sau cu o linie de subliniere.
- ☑ C++ consideră minusculele și majusculele ca fiind distincte.
- ☑ Tipul unei variabile determină tipul valorilor pe care acea variabilă le poate reține. Tipurile uzuale de variabile sunt *char*, *int*, *float* și *long*.
- ☑ Comentariile sporesc lizibilitatea programului prin explicațiile despre operațiile efectuate. Programele C++ marchează comentariile prin două semne slash (//).

Lecția 6

Efectuarea de operații elementare

În lecția 5, „Programele păstrează informații în variabile", ați învățat să declarați și să folosiți variabile în cadrul programelor. Pe măsură ce programele vor crește în complexitate, asupra valorilor conținute în variabile veți efectua diferite operații aritmetice, precum adunarea, scăderea, înmulțirea și împărțirea. Lecția de față vă arată cum puteți folosi operatorii aritmetici din C++ pentru a efectua astfel de operații. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru efectuarea de operații matematice în cadrul programelor, veți folosi operatorii aritmetici din C++.
- Pentru a asigura evaluarea operațiilor aritmetice într-o manieră consistentă, C++ atribuie fiecărui operator o anume prioritate.
- Folosind paranteze în cadrul expresiilor matematice, puteți controla ordinea în care C++ va efectua fiecare operație.
- Multe programe C++ adaugă sau scad valoarea unu din variabile prin intermediul operatorilor de incrementare (++) și decrementare (--).

După ce veți învăța să recunoașteți diferiții operatori aritmetici din C++, veți găsi efectuarea operațiilor matematice ca fiind foarte ușoară!

Operatori matematici de bază

Indiferent de scopul urmărit, majoritatea programelor C++ vor efectua adunări, scăderi, înmulțiri sau împărțiri de valori. Așa cum veți vedea, programele pot efectua operații aritmetice cu constante (cum ar fi 3 * 5) sau cu variabile (cum ar fi *plata – total*). Tabelul 6.1 enumeră operatorii matematici de bază din C++.

Operator	Scop	Exemplu
+	Adunare	total = cost + taxa;
	Scădere	rest = plata – total;
*	Înmulțire	taxa = cost * procent_taxa;
/	Împărțire	medie = total / numar;

Tabelul 6.1 *Operatorii matematici de bază din C++.*

Programul următor, *ShowMath.CPP*, foloseşte *cout* pentru a afişa rezultatele mai multor operaţii aritmetice simple:

```
#include <iostream.h>

void main(void)
  {
    cout << "5 + 7 = " << 5 + 7 << endl;
    cout << "12 - 7 = " << 12 - 7 << endl;
    cout << "1.2345 * 2 = " << 1.2345 * 2 << endl;
    cout << "15 / 3 = " << 15 / 3 << endl;
  }
```

Priviţi cu atenţie instrucţiunile programului. Observaţi că fiecare expresie apare mai întâi între ghilimele, ceea ce determină programul să afişeze pe ecran caracterele respective (precum 5 + 7 =). Apoi, programul afişează rezultatul operaţiei, urmat de o linie nouă (*endl*). Atunci când compilaţi şi rulaţi acest program, pe ecran vor fi afişate următoarele:

```
C:\> ShowMath <Enter>
5 + 7 = 12
12 - 7 = 5
1.2345 * 2 = 2.469
15 / 3 = 5
```

Programul *ShowMath* efectuează operaţii aritmetice folosind exclusiv valori constante. Programul următor, *MathVars.CPP*, efectuează operaţii aritmetice folosind variabile:

```
#include <iostream.h>

void main(void)
  {
    float cost = 15.50; // Costul unui obiect
    float sales_tax = 0.06; // Taxa pe valoare adaugata
                            // este 6%
    float amount_paid = 20.00; // Suma platita de
                               // cumparator
    float tax, change, total; // Taxa pe valoare adaugata,
                              // restul pentru cumparator
                              // si totalul de plata

    tax = cost * sales_tax;
    total = cost + tax;
```

```
    change = amount_paid - total;
    cout << "Item Cost: $" << cost << "\tTax: $" << tax <<
      "\tTotal: $" << total << endl;

    cout << "Customer change: $" << change << endl;
}
```

În acest caz, programul folosește exclusiv variabile în virgulă mobilă. Precum vedeți, programul atribuie valori variabilelor odată cu declararea acestora. Apoi sunt efectuate operații aritmetice asupra variabilelor pentru a determina taxa pe valoare adăugată, costul total al mărfii și restul cuvenit clientului. La compilarea și rularea acestui program, pe ecran vor fi afișate următoarele:

```
C:\> MathVars <Enter>
Item Cost: $15.5          Tax: $0.93 Total: $16.43
Customer change: $3.57
```

Luați-vă acum câteva minute pentru a modifica programul, schimbând valorile variabilelor. Ați putea, de pildă, să atribuiți variabilei *cost* valoarea 25.00 și variabilei *amount_paid* valoarea 100.00.

Incrementarea valorii unei variabile cu 1

În programare, o operație uzuală pe care o veți efectua este adunarea valorii 1 la o variabilă întreagă. De exemplu, să presupunem că un program folosește o variabilă numită *count* pentru a urmări numărul de fișiere pe care le-a tipărit. De fiecare dată când programul tipărește un fișier, el va aduna 1 la valoarea curentă din *count*. Programul poate incrementa valoarea *count* prin intermediul operatorului de atribuire C++, ca mai jos:

```
count = count + 1;
```

În acest caz, programul obține mai întâi valoarea *count* și apoi adună 1 la acea valoare. Rezultatul adunării este apoi depus de către program înapoi în variabila *count*. Programul care urmează, *IncCount.CPP*, folosește operatorul de atribuire pentru a incrementa variabila *count* (care conține inițial valoarea 1000) cu 1 (atribuind rezultatul 1001 aceleiași variabile):

```
#include <iostream.h>

void main(void)
  {
    int count = 1000;
    cout << "count's starting value is " << count << endl;
    count = count + 1;
    cout << "count's ending value is " << count << endl;
  }
```

După compilarea şi rularea programului *IncCount.CPP*, pe ecran vor fi afişate următoarele:

```
C:\> IncCount <Enter>
count's starting value is 1000
count's ending value is 1001
```

Deoarece incrementarea valorii unei variabile este o operaţie frecventă în cadrul programelor, C++ pune la dispoziţie un *operator de incrementare*, simbolizat de două semne plus (++). Operatorul de incrementare reprezintă o scurtătură pentru adunarea valorii 1 la o variabilă. Instrucţiunile următoare, de pildă, incrementează ambele valoarea *count* cu 1:

```
count = count + 1;          count++;
```

Programul următor, *Inc_Op.CPP*, foloseşte operatorul de incrementare pentru a incrementa cu 1 valoarea *count*:

```
#include <iostream.h>

void main(void)
  {
    int count = 1000;
    cout << "count's starting value is " << count << endl;
    count++;
    cout << "count's ending value is " << count << endl;
  }
```

Programul *Inc_Op.CPP* funcţionează identic cu *IncCount.CPP*, care folosea operatorul de atribuire pentru incrementarea valorii variabilei. Atunci când întâlneşte un operator de incrementare, C++ citeşte mai întâi valoarea variabilei, adună 1 la acea valoare şi apoi depune rezultatul înapoi în variabilă.

Despre operatorii de incrementare prefixat (înainte) şi postfixat (după)

Atunci când folosiţi operatorul de incrementare, acesta poate fi plasat înainte sau după variabilă, aşa cum se vede aici:

```
++variabila;          variabila++;
```

În primul caz, operatorul apare în faţa variabilei, ceea ce îl face un *operator de incrementare prefixat*. În cel de al doilea caz, operatorul apare după variabilă, fiind un *operator de incrementare postfixat*. Când programaţi trebuie să ştiţi că C++ tratează cei doi operatori în mod diferit. Spre exemplu, fie următoarea instrucţiune de atribuire:

```
current_count = count++;
```

Instrucțiunea de atribuire instruiește C++ să atribuie valoarea *count* curentă variabilei *current_count*. În plus, operatorul de incrementare determină C++ să incrementeze apoi valoarea curentă *count*. Utilizarea operatorului postfixat în cazul nostru face ca instrucțiunea de mai sus să fie echivalentă cu următoarele două instrucțiuni:

```
current_count = count;
count = count + 1;
```

Să luăm acum următoarea instrucțiune de atribuire care folosește operatorul de incrementare prefixat:

```
current_count = ++count;
```

În acest caz, operatorul prefixat determină C++ să incrementeze *mai întâi* valoarea *count* și *apoi* să atribuie rezultatul variabilei *current_count*. Utilizarea operatorului de incrementare prefixat face ca instrucțiunea de mai sus să fie echivalentă cu următoarele două instrucțiuni:

```
count = count + 1;
current_count = count;
```

Este important să înțelegeți operatorii de incrementare prefixat și postfixat deoarece îi veți întâlni în majoritatea programelor C++. Programul următor, *Pre_Post.CPP*, ilustrează utilizarea operatorilor de incrementare prefixat și postfixat:

```
#include <iostream.h>

void main(void)
  {
    int small_count = 0;
    int big_count = 1000;

    cout << "small_count is " << small_count << endl;
    cout << "small_count++ yields " << small_count++
      << endl;
    cout << "small_count ending value " << small_count
      << endl;

    cout << "big_count is " << big_count << endl;
```

```
      cout << "++big_count yields " << ++big_count << endl;
      cout << "big_count ending value " << big_count << endl;
   }
```

După compilarea şi rularea programului *Pre_Post.CPP*, pe ecran vor fi afişate următoarele:

```
C:\> Pre_Post <Enter>
small_count is 0
small_count++ yields 0
small_count ending value 1
big_count is 1000
++big_count yields 1001
big_count ending value 1001
```

Programul *Pre_Post.CPP* foloseşte operatorul de incrementare postfixat asupra variabilei *small_count*. Ca rezultat, programul afişează valoarea curentă a variabilei (0) şi apoi incrementează valoarea cu 1. Asupra variabilei *big_count* este aplicat operatorul de incrementare prefixat. În consecinţă, programul incrementează mai întâi valoarea variabilei (1000 + 1) şi apoi afişează rezultatul (1001). Opriţi-vă puţin pentru a edita acest program, înlocuind primul operator postfixat cu un operator prefixat şi cel de al doilea operator prefixat cu unul postfixat. Compilaţi şi rulaţi programul, observând cum modificarea operatorilor schimbă rezultatele programului.

C++ oferă şi un operator de decrementare

Aşa cum tocmai aţi aflat, două semne plus (++) reprezintă operatorul de incrementare din C++. În mod similar, două semne minus (–) reprezintă *operatorul de decrementare* din C++, care decrementează valoarea unei variabile cu 1. Ca şi în cazul operatorului de incrementare, C++ acceptă un operator de decrementare prefixat şi unul postfixat. Programul următor, *DecCount.CPP*, ilustrează modul de utilizare al operatorului de decrementare din C++:

```
#include <iostream.h>

void main(void)
   {
      int small_count = 0;
      int big_count = 1000;

      cout << "small_count is " << small_count << endl;
      cout << "small_count-- yields " << small_count-- << endl;
      cout << "small_count ending value " << small_count << endl;

      cout << "big_count is " << big_count << endl;
```

```
    cout << "--big_count yields " << --big_count << endl;
    cout << "big_count ending value " << big_count << endl;
}
```

La compilarea și rularea programului *DecCount.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> DecCount <Enter>
small_count is 0
small_count-- yields 0
small_count ending value is -1
big_count is 1000
--big_count yields 999
big_count ending value is 999
```

Precum vedeți, operatorii de decrementare prefixat și postfixat din C++ funcționează exact ca operatorii de incrementare corespunzători, cu diferența că aceștia decrementează valoarea variabilei cu 1.

Alți operatori din C++

Această lecție s-a concentrat asupra operatorilor aritmetici de bază din C++ și asupra operatorilor de incrementare și decrementare. Inspectând programe C++ ați putea întâlni, de asemenea, unul sau mai mulți dintre operatorii prezenți în tabelul 6.2.

Operator	Funcție
%	Operatorul modulo; întoarce restul unei împărțiri cu întregi
~	Operatorul de complementare față de unu; inversează biții unei valori
&	Operator *ȘI* la nivel de bit; efectuează operația *ȘI* între biții corespunzători ai celor două valori
|	Operator *SAU* la nivel de bit; efectuează operația *SAU* între biții corespunzători ai celor două valori
^	*SAU* exclusiv la nivel de bit; efectuează operația *SAU* exclusiv între biții corespunzători ai celor două valori
<<	Deplasare la stânga la nivel de bit; deplasează la stânga biții unei valori cu numărul de poziții specificate
>>	Deplasare la dreapta la nivel de bit; deplasează la dreapta biții unei valori cu numărul de poziții specificate

Tabelul 6.2 *Operatorii C++ pe care i-ați putea întâlni.*

Despre prioritățile operatorilor

Atunci când implementați operații aritmetice în C++ trebuie să țineți cont de faptul că C++ efectuează operațiile într-o anumită ordine, în funcție de *prioritățile operatorilor* – adică C++ consideră că unii operatori sunt mai importanți decât alții și va efectua mai întâi operațiile mai importante. De exemplu, pe baza priorităților operatorilor, C++ va efectua o înmulțire înaintea unei adunări. Pentru a înțelege mai bine prioritățile operatorilor, să luăm expresia următoare:

```
rezultat = 5 + 2 * 3;
```

În funcție de ordinea în care C++ ar efectua operațiile de înmulțire și de adunare, rezultatele obținute diferă, ca mai jos:

```
rezultat = 5 + 2 * 3;
         = 7 * 3;
         = 21;
```

```
rezultat = 5 + 2 * 3;
         = 5 + 6;
         = 11;
```

Pentru a evita orice confuzii, C++ asociază fiecărui operator o prioritate care determină ordinea în care vor fi efectuate operațiile. Pentru că C++ efectuează operațiile într-o ordine bine precizată, programele vor efectua calculele matematice într-o manieră consistentă.

Tabelul 6.3 enumeră prioritățile operatorilor din C++. Operatorii care apar în secțiunea superioară au cea mai mare prioritate. În cadrul fiecărei secțiuni, operatorii au aceeași prioritate. Dacă examinați tabelul, puteți vedea că C++ atribuie o prioritate mai mare înmulțirii decât adunării. Mulți dintre operatorii care apar vă sunt necunoscuți. Nu vă faceți probleme deocamdată despre acești operatori. La momentul când veți fi parcurs cartea veți fi folosit (și înțeles) fiecare operator.

Operator	Nume	Exemplu
	Rezoluție de scop	nume_clasa::nume_membru_clasa
	Rezoluție globală	::nume_variabila
	Selector de membru	obiect.nume_membru
->	Selector de membru	pointer->nume_membru
[]	Indice	pointer[element]
()	Apel de funcție	expresie(parametri)
()	Constructor cu valori	tip(parametri)
sizeof	Dimensiunea unui obiect	sizeof expresie
sizeof	Dimensiunea unui tip	sizeof(tip)
++	Incrementare postfixată	variabila++

Tabelul 6.3 *Prioritățile operatorilor din C++.*

Operator	Nume	Exemplu
++	Incrementare prefixată	++variabila
	Decrementare postfixată	variabila--
	Decrementare prefixată	--variabila
&	Operator de adresare	&variabila
*	Operator de indirectare	*pointer
new	Operator de alocare	new tip
delete	Operator de dezalocare	delete pointer
delete[]	Dezalocare de vector	delete pointer
~	Complement față de unu	~expresie
!	Operatorul NON	! expresie
+	Plus unar	+1
-	Minus unar	-1
()	Operator de conversie	(tip) expresie
.*	Selector de membru	obiect.*pointer
->	Selector de membru	obiect->*pointer
*	Înmulțire	expresie * expresie
/	Împărțire	expresie / expresie
%	Modulo	expresie % expresie
+	Adunare	expresie + expresie
-	Scădere	expresie - expresie

Tabelul 6.3 *Prioritățile operatorilor din C++. (continuare)*

Determinarea ordinii în care C++ efectuează operații

După cum ați aflat, C++ atribuie operatorilor diferite priorități care determină ordinea în care sunt efectuate operațiile. Din păcate, pot exista situații când ordinea în care C++ efectuează operațiile aritmetice nu este și ordinea pe care o doriți. De exemplu, să presupunem că un program trebuie să adune două prețuri și apoi să înmulțească rezultatul cu o taxă procentuală, ca mai jos:

```
cost = pret_a + pret_b * 1.06;
```

Din păcate, în acest caz C++ va efectua mai întâi înmulțirea (pret_b * 1.06) și apoi va aduna valoarea *pret_a*.

Dacă un program trebuie să efectueze operații aritmetice într-o anumită ordine, atunci puteți plasa expresiile între paranteze. La evaluarea expresiilor, C++ efectuează întot-

deauna mai întâi operațiile din program care sunt grupate în paranteze. Să luăm, de exemplu, expresia următoare:

```
rezultat = (2 + 3) * (3 + 4);
```

C++ va evalua această expresie în următoarea manieră:

```
rezultat = (2 + 3) * (3 + 4);
         = (5) * (3 + 4);
         = 5 * (7);
         = 5 * 7;
         = 35;
```

Prin această grupare a expresiilor cu paranteze, puteți determina ordinea în care C++ va efectua operațiile aritmetice. Revenind la exemplul precedent, programul poate aduna cele două prețuri într-o paranteză, ca mai jos:

```
cost = (pret_a + pret_b) * 1.06;
```

Țineți cont de depășire la operațiile aritmetice

În lecția 5 ați văzut că atunci când atribuiți unei variabile o valoare care se află în afara intervalului de valori pe care le permite tipul acelei variabile se produce o eroare de depășire. La efectuarea operațiilor aritmetice trebuie să țineți cont de aceste erori de depășire. De exemplu, programul următor, *MathOver.CPP*, înmulțește 200 cu 300 și atribuie rezultatul unei variabile de tip *int*. Cum rezultatul înmulțirii (60.000) depășește, însă, valoarea maximă admisă pentru o variabilă de tip *int* (32.767), se produce o eroare de depășire:

```
#include <iostream.h>

void main(void)
  {
    int result;

    result = 200 * 300;

    cout << "200 * 300 = " << result << endl;
  }
```

După compilarea și rularea programului *MathOver.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> MathOver <Enter>
200 * 300 = -5536
```

Ce trebuie să ştiţi

În lecţia de faţă ne-am oprit asupra operatorilor aritmetici de bază şi asupra operatorilor de incrementare din C++. Aşa cum aţi aflat, pentru a asigura o modalitate bine definită de efectuare a operaţiilor matematice, C++ atribuie fiecărui operator câte o prioritate care determină ordinea în care C++ efectuează fiecare operaţie. În lecţia 7, „Citirea de informaţii de la tastatură", veţi învăţa să folosiţi un flux de intrare numit *cin* pentru a efectua operaţii de intrare de la tastatură. Dar înainte de a trece la lecţia 7, asiguraţi-vă că aţi reţinut următoarele aspecte importante:

☑ C++ foloseşte operatorii +, -, * şi / pentru adunare, scădere, înmulţire şi împărţire.

☑ C++ oferă operatorii de incrementare prefixat (înainte) şi postfixat (după) pentru a aduna 1 la valoarea unei variabile.

☑ C++ oferă operatorii de decrementare prefixat (înainte) şi postfixat (după) pentru a scădea 1 din valoarea unei variabile.

☑ Operatorii prefixaţi (înainte) determină C++ ca mai întâi să incrementeze (decrementeze) valoarea variabilei şi apoi să folosească valoarea respectivă.

☑ Operatorii postfixaţi (după) determină C++ să folosească mai întâi valoarea variabilei şi apoi să incrementeze (decrementeze) valoarea respectivă.

☑ Pentru a asigura o evaluare bine definită a expresiilor, C++ atribuie fiecărui operator câte o prioritate care determină ordinea de efectuare a operaţiilor.

☑ Atunci când vreţi să interveniţi asupra ordinii în care sunt efectuate operaţiile aritmetice, plasaţi expresiile între paranteze. C++ evaluează întotdeauna mai întâi expresiile din paranteze.

Lecția 7

Citirea de informații de la tastatură

Pe parcursul acestei cărți, programele prezentate au folosit intens fluxul de ieșire *cout* pentru a afișa rezultate pe ecran. În lecția de față veți vedea că C++ pune la dispoziție și un flux de intrare, numit *cin*, prin intermediul căruia programele pot citi informații introduse de utilizator. Atunci când folosiți *cin* pentru a citi date de la tastatură, trebuie să specificați una sau mai multe variabile cărora *cin* le va atribui valorile de intrare. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Programele C++ pot folosi fluxul de intrare *cin* pentru a citi și atribui unor variabile literele și numerele pe care utilizatorul le introduce prin tastatură.
- După ce folosiți *cin* pentru a citi și atribui unei variabile informații de la tastatură, conținutul acelei variabile poate fi folosit ca și cum programul ar fi stocat în variabilă valoarea respectivă prin intermediul operatorului de atribuire.
- Atunci când utilizează *cin* pentru a citi date de la tastatură, programele trebuie să țină cont de erori de depășire și de erorile care apar atunci când utilizatorul introduce o valoare de alt tip (o valoare care nu se potrivește cu tipul de valoare care poate fi stocat în variabila în cauză).

Așa cum ați văzut, atunci când folosesc fluxul de ieșire *cout*, programele plasează datele în cadrul fluxului cu ajutorul operatorului de *inserare* (<<). Într-un mod similar, atunci când folosesc fluxul de intrare *cin* pentru a citi informații de la tastatură, programele apelează la operatorul de *extragere* (>>).

Primii pași cu cin

Așa cum fluxul de ieșire *cout* permitea programelor afișarea de date pe ecranul monitorului, fluxul de intrare *cin* permite unui program să citească date de la tastatură. Atunci când utilizează *cin* pentru a citi date de la tastatură, programele trebuie să precizeze o variabilă în care *cin* va plasa datele respective. Programul următor, *FirstCin.CPP*, folosește *cin* pentru a citi un număr pe care îl tastează o persoană. Programul atribuie numărul tastat de utilizator unei variabile numite *number* și apoi afișează valoarea acestei variabile prin intermediul fluxului de ieșire *cout*, așa cum este ilustrat în continuare:

```
#include <iostream.h>

void main(void)
   {
     int number; // Numarul citit de la tastatura

     cout << "Type your favorite number and press Enter:
     cin >> number;
     cout << "Your favorite number is " << number << endl;
   }
```

Atunci când compilați și rulați programul *FirstCin.CPP*, pe ecran va apărea un mesaj care vă solicită să tastați numărul dumneavoastră preferat. După ce tastați numărul și apăsați Enter, programul va atribui acel număr variabilei *number*. Cu ajutorul *cout*, programul va afișa apoi un mesaj care conține numărul dumneavoastră favorit.

Programul următor, *TwoNbrs.CPP*, vă solicită introducerea a două numere. Programul atribuie numerele respective variabilelor *first* și *second*, iar apoi afișează numerele prin intermediul *cout*, ca mai jos:

```
#include <iostream.h>

void main(void)
   {
     int first, second; // Numere introduse de la tastatura

     cout << "Type two numbers and press Enter: ";
     cin >> first >> second;
     cout << "The numbers typed were " << first << " and "
       << second << endl;
   }
```

Remarcați prezența celor doi operatori de extragere în cadrul *cin*:

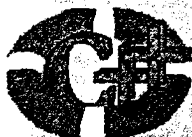
```
cin >> first >> second;
```

În acest caz, *cin* va atribui prima valoare introdusă variabilei *first* și cea de a doua valoare variabilei *second*. Dacă programul ar necesita și o a treia valoare, puteți să folosiți un al treilea operator de extragere, ca mai jos:

```
cin >> first >> second >> third;
```

Atunci când folosiți *cin* pentru a citi caractere de la tastatură, fluxul *cin* va folosi primul *caracter de spațiere* (spațiu, tabulator sau retur de car) pentru a determina unde se termină o valoare și începe următoarea. Experimentați cu programul *TwoNbrs*, separând numerele cu un spațiu, un tabulator sau un retur de car. De exemplu, la prima rulare a programului apăsați între numere tasta Tab. A doua oară când rulați programul, apăsați Enter după fiecare număr.

Citirea de informații de la tastatură prin intermediul cin



Atunci când un program trebuie să citească informații de la tastatură, acesta poate apela la fluxul de intrare *cin*. La utilizarea fluxului *cin* va trebui să folosiți operatorul de extragere (>>) pentru a specifica variabila în care doriți ca *cin* să plaseze datele, așa cum se vede în continuare:

```
cin >> o_variabila;
```

Operatorul de extragere este numit astfel deoarece este un operator care extrage (preia) datele dintr-un flux de intrare și atribuie datele variabilei care apare la dreapta operatorului.

Țineți cont de erorile de depășire

Dacă programele citesc date prin intermediul *cin*, va trebui să țineți cont de erorile ce pot apărea atunci când un utilizator tastează un număr nepotrivit. Spre exemplu, rulați programul *FirstCin* pe care tocmai l-ați creat. Atunci când programul vă solicită introducerea numărului favorit, tastați numărul 1000000 și apăsați Enter. Programul nu va afișa numărul 1000000 ca fiind valoarea introdusă. În schimb, pentru că 1000000 depășește valoarea maximă pe care o permite tipul *int*, va apărea o eroare de depășire.

Dacă studiați programul *FirstCin.CPP* veți observa că *cin* atribuie numărul introdus unei variabile de tip *int*. Așa cum ați învățat în lecția 5, „Programele păstrează informații în variabile“, variabilele de tip *int* pot reține valori aflate între -32.768 și 32.767. Deoarece o variabilă de tip *int* nu poate păstra valoarea 1000000, programul a generat o eroare de depășire. Rulați programul încă de câteva ori, tastând numere negative și pozitive. Remarcați că erorile apar atunci când depășiți intervalul de valori admise pentru variabila în care *cin* plasează datele citite.

Țineți cont de erorile de nepotrivire de tip

Așa cum am văzut, programul *FirstCin.CPP* se așteaptă ca utilizatorul să introducă o valoare aflată între -32.768 și 32.767. Dacă în loc să introducă o valoare din afara acestui interval utilizatorul tastează caractere sau alte simboluri, atunci se va produce o eroare de *nepotrivire de tip*. Cu alte cuvinte, programul așteaptă o valoare de un anumit tip (*int*), iar utilizatorul a introdus o valoare de un alt tip (*char*). Pentru a exemplifica, rulați programul încă o dată. Atunci când sunteți solicitat să introduceți un număr, tastați literele *ABC*. Ca

mai devreme, deoarece programul așteaptă o valoare numerică întreagă și nu litere, se va produce o eroare.

Efectuați experiențe similare cu programul *TwoNbrs*, tastând eventual valori fără sens sau chiar numere cu virgulă. Așa cum veți descoperi, atunci când tastați valori eronate, ieșirea programului conține la rândul său erori. În lecții ulterioare veți învăța cum puteți efectua operații de citire astfel încât să reduceți la minim posibilitatea de apariție a unor astfel de erori. Deocamdată, însă, este suficient să rețineți că este posibil să apară aceste erori.

Citirea caracterelor

Ambele programe anterioare au folosit *cin* pentru a citi numere întregi în variabile de tip *int*. Programul care urmează, *Cin_Char.CPP*, utilizează fluxul de intrare *cin* pentru a citi de la tastatură un caracter. După cum puteți vedea, programul citește caracterul într-o variabilă de tip *char*:

```
#include <iostream.h>

void main(void)
   {
     char letter;

     cout << "Type any character and press Enter: ";
     cin >> letter;
     cout << "The letter typed was " << letter << endl;
   }
```

Compilați și experimentați programul *Cin_Char.CPP*, tastând eventual mai multe caractere și urmărind reacția programului. Așa cum veți vedea, programul nu lucrează decât cu un singur caracter odată.

Citirea de valori de la tastatură

În partea a doua a acestei cărți veți învăța să plasați într-o variabilă cuvinte sau chiar o întreagă linie de text. Atunci veți învăța să folosiți fluxul de intrare *cin* pentru a citi cuvinte și linii întregi. Deocamdată, însă, ați putea dori să creați propriile programe simple care să citească valori de tip *float* sau *long*. Spre exemplu, programul următor, *Cin_Long.CPP*, folosește *cin* pentru a citi o valoare *long*:

```
#include <iostream.h>

void main(void)
   {
     long value;
```

```
    cout << "Type a large number and press Enter: ";
    cin >> value;
    cout << "The number you typed was " << value << endl;
}
```

Ca şi mai devreme, experimentaţi cu programul *Cin_Long.CPP* tastând numere mari (şi negative).

Redirectarea I/E şi fluxul de intrare cin



Aşa cum aţi văzut în lecţia 4, „Afişarea mesajelor pe ecran", atunci când programele folosesc fluxul de ieşire *cout* este posibil ca un utilizator să redirecteze ieşirea programului de la ecran către un fişier sau o imprimantă. După cum am discutat, fluxul de ieşire *cout* corespunde ieşirii standard a sistemului de operare. Într-un mod similar, fluxul de intrare *cin* corespunde intrării standard a sistemului de operare. În consecinţă, atunci când un program foloseşte *cin* pentru a efectua operaţii de citire, utilizatorul poate redirecta intrarea programului de la tastatură la un fişier. În lecţii ulterioare veţi afla cum puteţi să scrieţi programe care citesc şi prelucrează intrarea redirectată.

Ce trebuie să ştiţi

În lecţia de faţă aţi învăţat să folosiţi fluxul de intrare *cin* pentru a efectua operaţii de citire de la tastatură. După cum aţi văzut, atunci când un program apelează la *cin* pentru a citi date de la tastatură, trebuie să specificaţi variabilele cărora *cin* le va atribui valorile pe care le tastează utilizatorul. În lecţia 8, „Învăţaţi programul să ia decizii", veţi învăţa să folosiţi instrucţiunea *if* din C++ pentru a da programelor posibilitatea de a lua decizii proprii. Dar înainte de a trece la lecţia 8, asiguraţi-vă că aţi reţinut următoarele aspecte importante:

- ☑ C++ pune la dispoziţie fluxul de intrare *cin* pe care programele îl pot utiliza pentru a citi date de la tastatură.
- ☑ Atunci când folosesc *cin* pentru a citi date, programele trebuie să specifice una sau mai multe variabile în care *cin* va plasa respectivele date.
- ☑ Pentru a atribui unei variabile o valoare citită trebuie să utilizaţi *cin* împreună cu operatorul de extragere (>>).
- ☑ Atunci când programele utilizează *cin* pentru a citi mai multe valori, *cin* foloseşte caracterele de spaţiere (spaţiu, tabulator sau retur de car) pentru a determina unde se termină o valoare şi începe următoarea.
- ☑ Dacă utilizatorul nu introduce tipul de date potrivit, este posibil să apară erori de depăşire sau de nepotrivire de tip, iar valorile pe care *cin* le va atribui variabilelor din program vor fi eronate.

Lecția 8

Învățați programul să ia decizii

Așa cum ați văzut, un program reprezintă o listă de instrucțiuni pe care calculatorul le execută pentru a îndeplini o sarcină anume. Toate programele C++ simple pe care le-ați creat până acum au început cu prima instrucțiune din program și au executat, în ordine, toate instrucțiunile până la sfârșitul programului. Pe măsură ce programele devin tot mai complexe, vor fi situații în care veți dori ca un program să execute o serie de instrucțiuni atunci când se îndeplinește o condiție și, eventual, o altă serie dacă acea condiție nu este îndeplinită. Cu alte cuvinte, veți vrea ca programele dumneavoastră să ia decizii și să reacționeze corespunzător. Lecția de față se oprește asupra instrucțiunii C++ *if* pe care programele o vor folosi pentru a lua astfel de decizii. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Programele C++ utilizează *operatorii relaționali* pentru a determina dacă două valori sunt egale sau dacă o valoare este mai mare sau mai mică decât cealaltă.
- Programele C++ folosesc instrucțiunea *if* pentru a lua decizii.
- Instrucțiunile C++ pot fi *simple* (o singură operație) sau *compuse* (mai multe operații pe care programul le grupează cu ajutorul acoladelor deschisă și închisă {}).
- Programele C++ utilizează instrucțiunea *if-else* pentru a executa o serie de instrucțiuni atunci când o condiție este adevărată și o altă serie de instrucțiuni atunci când respectiva condiție este falsă.
- Prin combinarea mai multor instrucțiuni *if-else*, programele pot testa și răspunde la diferite condiții.
- Prin intermediul operatorilor C++ *ŞI* (&&) și *SAU* (||), programele pot testa mai multe condiții, ca de pildă *Are utilizatorul un câine ŞI este câinele un dalmațian?*

Programele care iau decizii efectuează *prelucrări condiționale*. Cu alte cuvinte, programul va executa anumite instrucțiuni pe baza adevărului uneia sau mai multor condiții. Experimentați cu programele prezentate în această lecție. Colecția dumneavoastră de instrumente C++ devine acum suficient de mare pentru a vă permite crearea de programe utile.

Compararea a două valori

Pentru a lua decizii, programele trebuie să efectueze mai înainte un anumit test. De exemplu, un program ar putea testa dacă nota unui student la examen este egală cu 10, iar un alt program ar putea testa dacă prețul unui obiect este mai mare de $50,00. Pentru a efectua astfel de teste, programele folosesc operatorii relaționali din C++ prezentați în tabelul 8.1. *Operatorii relaționali* permit programelor să testeze în ce „relație" se află o valoare în raport cu o alta. Cu alte cuvinte, prin intermediul operatorilor relaționali

programele testează dacă o valoare este egală cu, mai mare decât sau mai mică decât o altă valoare. Atunci când programele utilizează operatorii relaţionali pentru a compara două valori, rezultatul comparaţiei este fie adevărat, fie fals – ceea ce înseamnă că cele două valori sunt fie egale (adevărat), fie diferite (fals). Toate instrucţiunile *if* din programele prezentate în această carte vor folosi operatorii relaţionali enumeraţi în tabelul 8.1.

Operator	Testează	Exemplu
==	Dacă două valori sunt egale	(nota == 10)
!=	Dacă două valori sunt diferite	(vechi != nou)
>	Dacă prima valoare este mai mare decât a doua	(pret > 50.00)
<	Dacă prima valoare este mai mică decât a doua	(salariu < 20000.00)
>=	Dacă prima valoare este mai mare sau egală cu a doua	(pret_actiune >= 30.0)
<=	Dacă prima valoare este mai mică sau egală cu a doua	(varsta <= 21)

Tabelul 8.1 *Operatorii relaţionali din C++.*

Primii paşi cu instrucţiunea if

Instrucţiunea *if* din C++ permite programelor efectuarea unui test şi executarea de instrucţiuni pe baza rezultatului testului. Formatul instrucţiunii *if* este următorul:

```
if (conditie_este_adevarata)
   instructiune;
```

Instrucţiunea *if* efectuează în mod normal testul cu ajutorul operatorilor relaţionali din C++. Dacă rezultatul testului este adevărat, programul execută instrucţiunea care urmează după *if*. În schimb, dacă rezultatul testului este fals, programul ignoră (sare) instrucţiunea care urmează. Programul următor, *First_If.CPP*, utilizează instrucţiunea *if* pentru a compara valoarea păstrată în variabila *test_score* cu valoarea 90. Dacă nota la test este mai mare sau egală cu 90, programul va afişa un mesaj care informează utilizatorul că a obţinut calificativul A. Altfel, dacă valoarea este mai mică decât 90, programul îşi încheie pur şi simplu execuţia:

```
#include <iostream.h>

void main(void)
  {
    int test_score = 95;
```

```
    if (test score >= 90)
      cout << "Congratulations, you got an A!" << endl;
}
```

După cum vedeți, pentru efectuarea testului programul folosește operatorul relațional mai-mare-sau-egal (>=) din C++. În cazul în care compararea valorilor duce la rezultatul adevărat, programul va executa instrucțiunea care urmează – în cazul nostru, afișarea mesajului prin intermediul *cout*. În cazul în care compararea duce la rezultatul fals, programul nu va afișa mesajul în cauză. Experimentați cu acest program, modificând nota la test la o valoare mai mică decât 90, și observați cum funcționează instrucțiunea *if*.

Despre instrucțiunile simple și compuse

Atunci când programele folosesc instrucțiunea *if* pentru prelucrări condiționale, vor exista situații în care un program va trebui să execute o singură instrucțiune atunci când condiția este adevărată. În alte cazuri, însă, programele vor trebui să execute mai multe instrucțiuni atunci când o condiție este adevărată. Dacă un program execută o singură instrucțiune ce urmează un *if*, respectiva instrucțiune este o *instrucțiune simplă*, ca mai jos:

```
if (test_score >= 90)
   cout << "Congratulations,
     you got an A!" << endl;
```

Instrucțiune simplă

Pentru ca un program să execute mai multe instrucțiuni atunci când o condiție este îndeplinită, va trebui să grupați acele instrucțiuni cu ajutorul acoladelor deschisă și închisă {}. Instrucțiunile care apar între acolade alcătuiesc o *instrucțiune compusă*, așa cum se vede aici:

```
if (test_score >= 90)
     {
        cout << "Congratulations, you got
          an A!" << endl;
        cout << "Your test score was "
          << test score << endl;
     }
```

Instrucțiune compusă

Nu este important să rețineți denumirile de instrucțiune simplă și compusă, ci să țineți minte că instrucțiunile înrudite trebuie grupate între acolade închisă și deschisă. Programul următor, *Compund.CPP*, modifică programul anterior pentru a afișa două mesaje în cazul în care nota la test este mai mare sau egală cu 90:

```
#include <iostream.h>

void main(void)
  {
    int test score = 95;

    if (test score >= 90)
      {
        cout << "Congratulations, you got an A!"
          << endl;
        cout << "Your test score was " << test score
          << endl;
      }
  }
```

Utilizarea instrucţiunilor simple şi compuse



Atunci când programele efectuează prelucrări condiţionale, vor fi cazuri în care acestea vor trebui să execute o singură instrucţiune (o instrucţiune simplă) dacă o condiţie este adevărată. În alte situaţii, însă, programele vor trebui să execute mai multe instrucţiuni (o instrucţiune compusă). Atunci când un program trebuie să execute două sau mai multe instrucţiuni înrudite pe baza unei condiţii, acele instrucţiuni trebuie grupate cu ajutorul acoladelor închisă şi deschisă, aşa cum se vede aici:

```
if (varsta >= 21)
  {
    cout << "Ai grija sa nu uiti sa votezi!" << endl;
    cout << "Ah, apropos, berea asta este
      pentru tine!" << endl;
  }
```

Precizarea de instrucţiuni alternative pentru condiţiile false

Cele două programe precedente foloseau o instrucţiune *if* pentru a determina dacă nota la test era mai mare sau egală cu 90. În cazul în care condiţia era adevărată, programele afişau pe ecran un mesaj. Atunci când condiţia era falsă, adică nota la test era mai mică decât 90, programele nu mai afişau nimic, ci pur şi simplu se încheiau. De cele mai multe ori, programele trebuie să specifice o serie de instrucţiuni care să fie executată atunci când o condiţie este adevărată şi o altă serie care să fie executată atunci când condiţia este falsă.

Pentru a preciza instrucțiunile de executat când condiția este falsă, programele trebuie să apeleze la instrucțiunea *else*. Formatul instrucțiunii *else* este următorul:

```
if (conditie este adevarata)
     instructiune;
else
     instructiune;
```

Programul următor, *If_Else.CPP*, folosește instrucțiunea *if* pentru a testa dacă nota la test este mai mare sau egală cu 90. În cazul în care condiția este adevărată, programul va afișa un mesaj de felicitare. În cazul în care condiția este falsă, programul va afișa un mesaj care sfătuiește studentul să lucreze mai mult, așa cum se vede aici:

```
#include <iostream.h>
void main(void)
  {
    int test_score = 95;
    if (test score >= 90)
      cout << "Congratulations, you got an A!" << endl;
    else
      cout << "You must work harder next time!" << endl;
  }
```

Instrucțiunile compuse sunt valabile și pentru else

După cum ați văzut, o instrucțiune compusă grupează instrucțiunile înrudite cu ajutorul acoladelor deschisă și închisă. Atunci când un program folosește o instrucțiune *else* pentru a specifica instrucțiunile care vor fi executate atunci când o condiție anume este falsă, puteți utiliza o instrucțiune compusă pentru a grupa mai multe instrucțiuni. Programul următor, *Cmp_Else.CPP*, utilizează o instrucțiune compusă atât pentru *if*, cât și pentru *else*:

```
#include <iostream.h>
void main(void)
  {
    int test_score = 65;
    if (test score >= 90)
      {
        cout << "Congratulations, you got an A!" << endl;
        cout << "Your test score was " << test_score
```

```
            << endl;
      }
   else
      {
         cout << "You should have worked harder!"
            << endl;
         cout << "You missed " << 100 - test_score
            << " points " << endl;
      }
   }
```

Ca şi mai devreme, petreceţi câteva minute experimentând cu acest program, modificând variabila *test_score* la valori mai mici sau mai mari decât 90. Programul următor, *GetScore.CPP*, foloseşte fluxul de intrare *cin* pentru a citi de la utilizator nota la test. Programul compară apoi nota cu 90, afişând mesaje corespunzătoare:

```
#include <iostream.h>

void main(void)
   {
   int test_score;
   cout << "Type in the test score and press Enter:
   cin >> test_score;
   if (test score >= 90)
      {
         cout << "Congratulations, you got an A!"
            << endl;
         cout << "Your test score was  << test_score
            << endl;
      }
   else
      {
         cout << "You should have worked harder!"
            << endl;
         cout << "You missed " << 100 - test_score <<
              points  << endl;
      }
   }
```

Precum vedeți, programul folosește fluxul de ieșire *cout* pentru a solicita utilizatorului să introducă nota de la test. Apoi, programul utilizează fluxul de intrare *cin* pentru a atribui răspunsul utilizatorului variabilei *test_score*. Compilați și rulați programul *GetScore.CPP*. Așa cum veți descoperi, prin combinarea operațiilor de citire și a procesării condiționale, programele pot deveni foarte puternice.

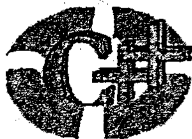
Utilizați indentarea pentru a spori lizibilitatea programelor

Dacă priviți programele prezentate în acest capitol, veți vedea că instrucțiunile care urmează un *if*, un *else* sau o acoladă deschisă sunt indentate. Printr-o astfel de indentare cu unul sau două spații a instrucțiunilor înlesniți determinarea modului în care sunt înrudite instrucțiunile pentru oricine citește programul, așa cum se vede în continuare:

```
if (test_score >= 90)
   {
     cout << "Congratulations, you got an A!"
       << endl;
     cout << "Your test score was  << test score
       << endl;
   }
else
   {
     cout << "You should have worked harder!"
       <<endl;
     cout << "You missed " << 100 - test_score <<
         points " << endl;
   }
```

În acest caz, prin simpla observare a indentării, un alt programator care vă citește codul poate identifica rapid ce instrucțiuni sunt legate de *if* și ce instrucțiuni sunt legate de *else*. Atunci când scrieți cod, folosiți o indentare similară pentru a face programele mai ușor de citit. C++ nu ține cont de indentare, dar programatorii care citesc și încearcă să înțeleagă codul o vor face.

Despre prelucrarea *if-else*



Pe măsură ce programele cresc în complexitate, acestea vor testa diferite condiții și vor executa o serie de instrucțiuni când o condiție este adevărată și o altă serie când condiția este falsă. Pentru a efectua o astfel de prelucrare condițională, programele vor apela la instrucțiunile *if-else*, așa cum se vede în continuare:

```
if (conditie_este_adevarata)
     instructiune;
else
     instructiune;
```

Dacă programele trebuie să execute mai multe instrucțiuni atunci când condiția este adevărată sau falsă, instrucțiunile înrudite trebuie grupate cu ajutorul acoladelor deschisă și închisă, ca mai jos:

```
if (conditie_este_adevarata)
   {
        prima_instructiune_pentru_adevarat;
        a_doua_instructiune_pentru_adevarat;
   }
else
   {
        prima_instructiune_pentru_fals;
        a_doua_instructiune_pentru_fals;
   }
```

Testarea a două sau mai multe condiții

Așa cum ați văzut, instrucțiunea *if* permite programului să testeze diverse condiții. Pe măsură ce programele vor spori în complexitate, vor fi situații în care veți testa mai multe condiții. Spre exemplu, un program ar putea testa dacă o notă este mai mare de 9 și dacă media curentă a unui elev este 10. Similar, ați putea testa dacă un utilizator deține un câine și dacă acel câine este un dalmațian. Pentru efectuarea unor asemenea operații se folosește operatorul logic *ȘI* (&&) din C++. În plus, în cazul în care doriți să testați dacă un utilizator are un câine sau o pisică, veți apela la operatorul logic *SAU* (||). Atunci când programele folosesc operatorii logici *ȘI* și *SAU* pentru a testa mai multe condiții, fiecare dintre condiții va trebui plasată între paranteze, ca mai jos:

```
if ((utilizator_are_un_caine) && (caine == dalmatian))
```

Întreaga condiție

După cum puteți vedea, programul plasează fiecare condiție între paranteze, iar toate acestea sunt încadrate de o pereche de paranteze exterioare.

```
if ((utilizator_are_un_caine) && (caine == dalmatian))
```

Atunci când într-un program se folosește operatorul logic *ŞI* (&&), pentru ca întreaga condiție să fie evaluată ca adevărată trebuie ca toate condițiile din cadrul instrucțiunii să fie adevărate. Dacă o singură condiție este falsă, atunci întreaga condiție devine falsă. De exemplu, dacă utilizatorul nu deține un câine, condiția de mai sus este falsă. Similar, în cazul în care câinele utilizatorului nu este un dalmațian, condiția este falsă. Pentru ca această condiție să fie adevărată, trebuie ca utilizatorul să aibă un câine și câinele să fie un dalmațian.

Instrucțiunea următoare folosește operatorul logic *SAU* (||) pentru a determina dacă utilizatorul deține un câine sau o pisică:

```
if ((utilizator_are_un_caine) || (utilizator_are_o_pisica))
```

Pentru ca o condiție ce folosește operatorul logic *SAU* să fie evaluată ca adevărată, este suficient ca una dintre condițiile incluse să fie adevărată. De exemplu, dacă utilizatorul are un câine, condiția de mai sus este adevărată. Dacă utilizatorul are o pisică, atunci condiția este din nou adevărată. De asemenea, condiția este adevărată și dacă utilizatorul are atât câine, cât și pisică. Singurul caz în care condiția este falsă este acela în care utilizatorul nu are nici câine, nici pisică.

C++ reprezintă valoarea adevărat printr-o valoare nenulă și valoarea fals prin zero

Multe programe C++ profită de faptul că C++ reprezintă valoarea adevărat prin orice valoare nenulă și valoarea fals prin zero. Spre exemplu, să presupunem că programul dumneavoastră folosește o variabilă numită *utilizator_are_un_caine* pentru a determina dacă utilizatorul are sau nu un câine. Dacă acesta nu are un câine, puteți atribui acestei variabile valoarea 0 (fals), ca mai jos:

```
utilizator_are_un_caine = 0;
```

Dacă utilizatorul deţine un câine, atunci puteţi atribui variabilei orice valoare nenulă, de pildă 1:

```
utilizator_are_un_caine = 1;
```

Programul poate testa apoi variabila prin intermediul unei instrucţiuni *if*, aşa cum se vede aici:

```
if (utilizator_are_un_caine)
```

Dacă variabila conţine o valoare nenulă, condiţia va fi evaluată ca adevărată; altfel, dacă variabila conţine valoarea 0, condiţia este falsă. Profitând de modul în care C++ reprezintă valorile de adevărat şi de fals, instrucţiunea anterioară este identică cu următoarea:

```
if (utilizator_are_un_caine == 1)
```

Programul care urmează, *Dog_Cat.CPP*, foloseşte variabilele *user_owns_a_dog* şi *user_owns_a_cat* în cadrul unei instrucţiuni *if* pentru a determina ce animale deţine utilizatorul:

```
#include <iostream.h>

void main(void)
  {
    int user_owns_a_dog = 1;
    int user_owns_a_cat = 0;

    if (user_owns_a_dog)
      cout << "Dogs are great" << endl;

    if (user_owns_a_cat)
      cout << "Cats are great" << endl;

    if ((user_owns_a_dog) && (user_owns_a_cat))
      cout << "Dogs and cats can get along" << endl;

    if ((user_owns_a_dog) || (user_owns_a_cat))
      cout << "Pets are great!" << endl;
  }
```

Experimentaţi cu acest program atribuind valoarea 1 ambelor variabile, apoi 0 ambelor variabile şi apoi 1 şi 0 pentru fiecare variabilă, respectiv. Cu ajutorul operatorilor logici *ŞI* şi *SAU*, testarea a două sau mai multe condiţii în cadrul programelor devine foarte uşoară.

Utilizarea operatorului NON din C++

După cum ați văzut, atunci când un program testează diverse condiții, există situații în care doriți ca programul să execute anumite instrucțiuni dacă o condiție este adevărată. În mod similar, ar putea exista situații în care doriți ca programul să execute o serie de instrucțiuni dacă o condiție nu este adevărată. Operatorul *NON* din C++, semnul exclamării (!), permite programului să testeze dacă o condiție *nu* este adevărată. De exemplu, instrucțiunea următoare testează dacă utilizatorul nu deține un câine:

```
if (! utilizator_are_un_caine)
   cout << "Ar trebui sa cumparati un caine" << endl;
```

Operatorul *NON* transformă o condiție falsă în adevărată și o condiție adevărată în una falsă. Spre exemplu, să presupunem că utilizatorul nu are un câine. Variabila *utilizator_are_un_caine* ar conține valoarea 0. Atunci când evaluează condiția împreună cu operatorul *NON*, C++ folosește valoarea curentă a variabilei (0) și aplică operatorul *NON*. Operatorul *NON* transformă valoarea 0 în 1 (adevărat). Întreaga condiție este apoi evaluată ca adevărată, iar programul execută instrucțiunile corespunzătoare.

Programul următor, *Use_Not.CPP*, ilustrează utilizarea operatorului *NON*:

```
#include <iostream.h>

void main(void)
   {
     int user_owns_a_dog = 0;
     int user_owns_a_cat = 1;

     if (! user_owns_a_dog)
        cout << "You should buy a dog" << endl;

     if (! user_owns_a_cat)
        cout << "You should buy a cat" << endl;
   }
```

Ca și mai înainte, experimentați cu diferite valori atribuite variabilelor *user_owns_a_dog* și *user_owns_a_cat* și observați rezultatele programului. Pe măsură ce programele devin mai complexe, acestea vor folosi în mod regulat operatorul *NON*. De exemplu, un program ar putea să continue efectuarea repetată a unei operații atât timp cât *nu* a ajuns la sfârșitul unui fișier.

Utilizarea operatorilor logici din C++



Atunci când specificați condiții în cadrul programelor, ar putea exista cazuri în care condițiile să aibă mai multe părți. De exemplu, un program ar putea testa dacă un angajat este plătit cu ora și dacă a lucrat 40 de ore în săptămâna curentă. Atunci când pentru ca o condiție să fie adevărată este necesar ca ambele sale părți să fie adevărate, folosiți operatorul logic *ŞI* (&&). Pentru utilizarea operatorului *ŞI*, plasați fiecare condiție între paranteze și ansamblul condițiilor între altă pereche de paranteze, ca mai jos:

```
if ((plata_angajat == pe_ora) && (ore_angajat > 40))
   instructiune;
```

Atunci când pentru ca o condiție să fie adevărată este necesar ca numai una din cele două părți ale sale să fie adevărată, programul ar trebui să folosească operatorul C++ *SAU* (||). Spre exemplu, condiția următoare testează dacă utilizatorul deține o mașină sau o motocicletă:

```
if ((vehicul == masina) || (vehicul == motocicleta))
   instructiune;
```

Ca și mai înainte, programul include fiecare condiție între paranteze. În unele cazuri, ați putea dori ca programul să execute o instrucțiune atunci când o condiție nu este adevărată. În asemenea situații, ar trebui să folosiți operatorul C++ *NON* (!). Operatorul *NON* transformă o condiție adevărată în una falsă și o condiție falsă în una adevărată. Operatorii C++ *ŞI*, *SAU* și *NON* se numesc *operatori logici*.

Tratarea unor condiții diferite

Programele din această lecție au folosit *if* și *else* pentru a indica o serie de instrucțiuni pe care programul o va executa atunci când o condiție este adevărată și o altă serie de instrucțiuni pe care programul o va executa atunci când condiția este falsă. Ar putea exista, însă, cazuri în care un program trebuie să testeze mai multe condiții înrudite. De exemplu, să presupunem că un program trebuie să determine calificativul unui elev la un test. În acest scop, programul trebuie să testeze dacă nota este mai mare sau egală cu 90, 80, 70, 60 și așa mai departe. Programul următor, *ShowGrad.CPP*, folosește o serie de instrucțiuni *if-else* pentru determinarea calificativului:

```
#include <iostream.h>

void main(void)
  {
    int test_score;

    cout << "Type in your test score and press Enter: ";
    cin >> test_score;

    if (test_score >= 90)
      cout << "You got an A!" << endl;
    else if (test_score >= 80)
      cout << "You got a B!" << endl;
    else if (test_score >= 70)
      cout << "You got a C" << endl;
    else if (test_score >= 60)
      cout << "Your grade was a D" << endl;
    else
      cout << "You failed the test" << endl;
  }
```

Odată cu executarea primei instrucțiuni *if*, programul testează dacă nota de la test este mai mare sau egală ca 90. Dacă da, programul va afișa un mesaj care informează utilizatorul că a primit calificativul A. Dacă nota nu este mai mare sau egală cu 90, programul execută un *else if* pentru a testa dacă nota este mai mare sau egală cu 80. Acest proces este repetat până când este determinat calificativul corect. Ca mai devreme, experimentați cu acest program, introducând diferite note. De asemenea, remarcați că în acest caz instrucțiunea finală *else* nu este un *if-else*. Dacă elevul nu a obținut A, B, C sau D, atunci acesta a picat testul. Din acest motiv nu mai are sens ca programul să efectueze un alt test.

Utilizarea instrucțiunii switch

Așa cum tocmai ați învățat, prin combinarea unei serii de instrucțiuni *if-else* un program poate testa mai multe condiții. În programul anterior ați folosit instrucțiunile *if-else* pentru a determina dacă nota la un test se afla într-un anumit interval. Pentru cazurile în care programele trebuie să testeze valori precise, acestea pot apela la instrucțiunea *switch* din C++.

Atunci când folosiți instrucțiunea *switch* trebuie să precizați o condiție și apoi una sau mai multe valori de caz pe care programul va încerca să le potrivească cu respectiva condiție. De exemplu, programul următor, *Switch.CPP*, folosește o instrucțiune *switch* pentru a afișa un mesaj în funcție de calificativul unui elev:

```
#include <iostream.h>

void main(void)
  {
    char grade = 'B';

    switch (grade) {
      case 'A': cout << "Congratulations on your A"
        << endl;
        break;
      case 'B': cout << "Not bad, a B is ok" << endl;
        break;
      case 'C': cout << "C's are only average" << endl;
        break;
      case 'D': cout << "D's are terrible" << endl;
        break;
      default: cout << "No excuses! Study harder!"
        << endl;
          break;
    }
  }
```

Instrucţiunea *switch* este alcătuită din două părţi. Prima parte a instrucţiunii *switch* precizează condiţia care apare după cuvântul cheie *switch*. Cea de a doua parte specifică valorile posibile de testat. Atunci când întâlneşte o instrucţiune *switch*, programul examinează mai întâi condiţia şi apoi încearcă să găsească o valoare identică printre cazurile posibile. Atunci când întâlneşte o coincidenţă, programul execută instrucţiunile corespunzătoare. În cazul programului de mai sus, condiţia este satisfăcută de cazul în care litera calificativului este B. Din această cauză, programul va afişa un mesaj care informează utilizatorul că B nu este un calificativ rău. Acordaţi-vă câteva minute pentru a experimenta cu acest program, modificând litera calificativului şi observând rezultatele corespunzătoare. Cazul *default* reprezintă un caz pentru „orice altceva" care va satisface orice condiţie.

Observaţi utilizarea instrucţiunii *break* în fiecare dintre cazurile prezente în programul anterior. Aşa cum se întâmplă de fapt, atunci când identifică o valoare care coincide cu cea a condiţiei din instrucţiunea *switch*, C++ ia în considerare şi cazurile care urmează celui coincident. Instrucţiunea *break* determină C++ să încheie instrucţiunea *switch* curentă şi să continue execuţia programului cu prima instrucţiune care urmează instrucţiunii *switch*. Dacă înlăturaţi instrucţiunile *break* din programul precedent, acesta va afişa un mesaj nu numai pentru cazul coincident, ci şi pentru fiecare dintre cazurile care urmează acestuia (deoarece C++ consideră toate cazurile adevărate după ce unul este găsit adevărat).

Ce trebuie să știți

În lecția de față ați învățat să folosiți instrucțiunea *if* din C++ pentru a efectua prelucrări condiționale, ceea ce permite programelor dumneavoastră să ia propriile lor decizii. Așa cum ați văzut, programele pot folosi instrucțiunea *if* pentru a executa o serie de instrucțiuni atunci când o condiție este adevărată și instrucțiunea *else* pentru a indica instrucțiunile pe care programul le va executa atunci când condiția este falsă. În lecția 9, „Repetarea uneia sau mai multor instrucțiuni", veți învăța să utilizați instrucțiunile iterative din C++ pentru a repeta anumite instrucțiuni de un număr dat de ori sau până când se produce o condiție anume. De exemplu, ați putea repeta de 100 de ori aceleași instrucțiuni pentru a aduna 100 de note ale elevilor la un test. Dar înainte de a trece la lecția 9, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Operatorii relaționali din C++ permit programelor să testeze dacă două valori sunt egale sau diferite sau dacă o valoare este mai mare sau mai mică decât o alta.

☑ Instrucțiunea *if* din C++ permite programelor să testeze o condiție și să execute una sau mai multe instrucțiuni în cazul în care acea condiție este adevărată.

☑ Instrucțiunea *else* din C++ permite programelor să indice una sau mai multe instrucțiuni de executat atunci când condiția testată de o instrucțiune *if* este falsă.

☑ C++ reprezintă valoarea adevărat prin orice valoare nenulă, iar valoarea fals prin 0.

☑ Operatorii logici *ȘI* (&&) și *SAU* (||) din C++ permit programelor să testeze mai multe condiții odată.

☑ Operatorul logic *NON*(!) din C++ permite programelor să testeze dacă o condiție nu este adevărată.

☑ Dacă un program trebuie să execute mai multe instrucțiuni pentru o ramură *if* sau *else*, atunci aceste instrucțiuni trebuie plasate între acolade {}.

☑ Indentați instrucțiunile programului pentru a ajuta programatorii care vă citesc codul să identifice rapid instrucțiunile înrudite.

☑ Atunci când trebuie să testeze dacă o condiție coincide cu anumite valori, programele pot apela la instrucțiunea *switch*.

☑ Atunci când un program găsește un caz coincident în cadrul unei instrucțiuni *switch*, C++ consideră de asemenea coincidente toate cazurile care urmează. Prin intermediul instrucțiunii *break* puteți determina C++ să încheie instrucțiunea *switch* și să continue execuția programului cu prima instrucțiune care urmează după instrucțiunea *switch*.

Lecția 9

Repetarea uneia sau mai multor instrucțiuni

În lecția 8, „Învățați programul să ia decizii", ați învățat să folosiți instrucțiunea *if* din C++ pentru a lua decizii în cadrul programelor. Legată îndeaproape de o astfel de luare a deciziilor în cadrul unui program este capacitatea de a repeta una sau mai multe instrucțiuni de un număr dat de ori sau până când este îndeplinită o condiție anume, cum ar fi întâlnirea de către program a unui sfârșit de fișier. În lecția de față veți folosi construcțiile de buclare (iterative) din C++ pentru a repeta execuția uneia sau a mai multor instrucțiuni. În funcție de operațiile efectuate de program, ați putea repeta o serie de instrucțiuni folosind bucle *for*, *while* sau *do while*. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Instrucțiunea *for* din C++ permite programelor să repete o serie de instrucțiuni de un număr dat de ori.
- Instrucțiunea *while* din C++ permite programelor să repete o serie de instrucțiuni cât timp o condiție dată este adevărată.
- Instrucțiunea *do while* din C++ permite programelor să execute cel puțin o dată o serie de instrucțiuni și, eventual, să repete apoi acea serie în funcție de o condiție dată.

Capacitatea de a repeta instrucțiuni este o facilitate de programare foarte puternică. Experimentați cu programele prezentate în această lecție. După terminarea ei veți dispune de o cantitate considerabilă de cunoștințe privind programarea în C++.

Repetarea instrucțiunilor de un număr dat de ori

Atunci când programați, una dintre cele mai uzuale operații pe care programele o vor efectua va fi repetarea a una sau mai multe instrucțiuni de un număr dat de ori. De exemplu, un program ar putea executa repetat aceleași instrucțiuni pentru a tipări cinci copii ale unui fișier. Similar, un alt program ar putea repeta de 30 ori o serie de instrucțiuni pentru a determina, de pildă, dacă cele 30 de tipuri de acțiuni deținute au crescut sau au scăzut în valoare. Instrucțiunea *for* din C++ face ca repetarea în program de un număr dat de ori a uneia sau a mai multor instrucțiuni să fie foarte facilă.

La folosirea instrucțiunii *for* (numită de multe ori *buclă for*), un program trebuie să precizeze o variabilă, numită *variabilă de control*, care va urmări numărul de execuții ale buclei. De pildă, bucla *for* următoare utilizează variabila *count* pentru a urmări de câte ori programul a executat bucla. În exemplul care urmează, bucla va fi executată de zece ori:

```
for (contor = 1; contor <= 10; count++)
     instructiune;
```

Instrucțiunea *for* este alcătuită din patru părți: o inițializare, o condiție de test, instrucțiunile care vor fi repetate și o incrementare. Mai întâi, instrucțiunea *count = 1;* atribuie variabilei de control o valoare inițială. Bucla *for* efectuează această inițializare o singură dată, atunci când începe execuția buclei. Apoi, bucla testează condiția *count <= 10*. Dacă această condiție este adevărată, bucla *for* va executa instrucțiunea care urmează. În cazul în care condiția este falsă, bucla se va încheia, iar programul își va continua execuția cu prima instrucțiune care urmează buclei. Când condiția este adevărată și bucla *for* execută instrucțiunea, în continuare va fi incrementată variabila *count* prin intermediul instrucțiunii *count++*. În fine, programul testează condiția *count <= 10*. Dacă această condiție rămâne adevărată, programul va executa instrucțiunile, iar procesul de incrementare și de testare a variabilei *count* se va repeta, așa cum este ilustrat aici:

```
for (count = 1; count <= 10; count++)
```

Inițializare **Test** **Incrementare**

Programul următor, *FirstFor.CPP*, folosește o buclă *for* pentru a afișa pe ecran numerele între 1 și 100:

```
#include <iostream.h>

void main(void)
  {
    int count;

    for (count = 1; count <= 100; count++)
      cout << count << ' ';
  }
```

După cum puteți vedea, bucla *for* inițializează variabila *count* cu valoarea 1. Apoi, bucla testează dacă valoarea *count* este mai mică sau egală cu 100. Dacă da, bucla *for* va executa instrucțiunea corespunzătoare și va incrementa *count*, repetând apoi testul. Experimentați cu acest program, înlocuind valoarea 100 cu 10, 20 sau chiar 5000.

Programul următor, *AskCount.CPP*, afișează un mesaj care solicită utilizatorul să introducă numărul la care bucla se va încheia. Programul afișează apoi numerele aflate între 1 și valoarea introdusă de utilizator:

```
#include <iostream.h>

void main(void)
  {
    int count;
    int ending_value;
```

```
    cout << "Type in the ending value and press Enter: ";
    cin >> ending_value;

    for (count = 0; count <= ending_value; count++)
      cout << count << ' ';
  }
```

Experimentaţi cu programul *AskCount.CPP*, introducând valori ca 10, 1 şi chiar 0. Dacă introduceţi valoarea 0 sau –1, bucla *for* nu va fi executată niciodată, deoarece condiţia *count* <= *ending_value* va eşua de la bun început. Nu uitaţi că dacă introduceţi o valoare din afara intervalului de valori admise pentru o variabilă de tip *int* se va produce o eroare de depăşire. De exemplu, rulaţi programul şi introduceţi valoarea 50000. Pentru că valoarea depăşeşte valoarea maximă pe care o poate păstra o variabilă de tip *int*, depăşirea va duce la o valoare negativă, ceea ce va inhiba execuţia buclei.

Buclele *for* din C++ acceptă instrucţiuni compuse

În lecţia 8 aţi învăţat că atunci când un program execută mai multe instrucţiuni în cadrul unei ramuri *if* sau *else*, acele instrucţiuni trebuie grupate prin acolade. Acelaşi lucru este valabil şi pentru mai multe instrucţiuni dintr-o buclă *for*. Programul următor, *Add1_100.CPP*, parcurge numerele de la 1 la 100, afişând fiecare număr şi adunându-l apoi la un total:

```
#include <iostream.h>

void main(void)
  {
    int count;
    int total = 0;

    for (count = 1; count <= 100; count++)
      {
        cout << "Adding " << count << " to " << total;
        total = total + count;
        cout << " yields " << total << endl;
      }
  }
```

Prin gruparea instrucţiunilor între acolade, bucla for poate executa mai multe instrucţiuni de fiecare dată (la fiecare *iteraţie* a buclei).

Modificarea incrementului pentru buclă

Până aici, toate buclele *for* din programele acestei lecții au incrementat variabila de control a buclei cu 1 pentru fiecare iterație. O buclă *for* nu limitează, însă, programele la incrementarea variabilei cu 1. Programul următor, *By_Fives.CPP*, afișează numerele de la 0 la 100 din cinci în cinci:

```
#include <iostream.h>

void main(void)
  {
    int count;

    for (count = 0; count <= 100; count += 5)
      cout << count << ' ';
  }
```

Atunci când compilați și executați programul *By_Fives.CPP*, pe ecran vor fi afișate numerele 0, 5, 10 și așa mai departe, până la 100. Remarcați instrucțiunea pe care bucla *for* o folosește pentru incrementarea variabilei *count*:

```
count += 5;
```

Când doriți să adunați o valoare la valoarea curentă a unei variabile și să atribuiți rezultatul aceleiași variabile, C++ permite efectuarea acestei operații în două feluri. Pe de o parte, presupunând că programul trebuie să adune valoarea 5 la variabila *count*, aceasta se poate scrie astfel:

```
count = count + 5;
```

Pe de altă parte, C++ vă permite utilizarea unei notații prescurtate prin care să adunați valoarea 5 la variabila *count*:

```
count += 5;
```

Deoarece este mai ușor de scris, programatorii folosesc frecvent această notație prescurtată în cadrul buclelor. Atunci când utilizați o buclă *for*, C++ nu vă obligă să numărați crescător. Programul următor, *Cnt_Down.CPP*, folosește o buclă *for* pentru a număra descrescător și afișa numerele de la 100 la 1:

```
#include <iostream.h>

void main(void)
  {
    int count;

    for (count = 100; count >= 1; count--)
      cout << count << ' ';
  }
```

Așa cum puteți vedea, bucla *for* inițializează variabila *count* la 100. Cu fiecare iterație, bucla decrementează valoarea variabilei cu 1. Atunci când variabila *count* ajunge la valoarea 0, bucla se încheie.

Aveți grijă la buclele infinite



Așa cum ați aflat, o buclă *for* oferă programelor o cale de a repeta instrucțiuni înrudite de un număr dat de ori. Prin intermediul unei variabile de control, bucla *for* numără efectiv iterațiile efectuate. Atunci când este îndeplinită condiția de încheiere, programul încetează a mai repeta instrucțiunile și își continuă execuția cu prima instrucțiune care urmează buclei *for*.

Din păcate, datorită erorilor din programe, există cazuri în care o buclă nu atinge niciodată condiția de încheiere și astfel iterează fără sfârșit (sau până când opriți programul). Asemenea bucle fără sfârșit se numesc *bucle infinite*. Cu alte cuvinte, este vorba despre bucle care nu se pot încheia. Următoarea instrucțiune *for*, de pildă, creează o buclă infinită:

```
for (contor = 0; contor < 100; alta_variabila++)
  // Instructiuni
```

Precum vedeți, bucla *for* utilizează ca variabilă de control variabila *count*. În partea de incrementare a buclei, însă, programul incrementează o altă variabilă. Prin urmare, bucla nu incrementează niciodată variabila *count* și aceasta nu va atinge niciodată o valoare mai mare sau egală cu 100. În acest fel, bucla devine o buclă infinită.

Este important să rețineți că buclele *for* nu sunt limitate la folosirea variabilelor de tip *int* ca variabile de control. Programul următor, *LoopVar.CPP*, de pildă, folosește într-o buclă o variabilă de tip *char* pentru a afișa literele alfabetului și într-o altă buclă o variabilă de tip *float* pentru a afișa numere în virgulă mobilă:

```
#include <iostream.h>

void main(void)
  {
    char letter;
    float value;

    for (letter = 'A'; letter <= 'Z'; letter++)
      cout << letter;

    cout << endl;

    for (value = 0.0; value <= 1.0; value += 0.1)
      cout << value << ' ';

    cout << endl;
  }
```

După compilarea și rularea programului *LoopVar.CPP*, pe ecran vor fi afișate următoarele:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
```

Iterarea de un număr dat de ori



Atunci când programați, una dintre cele mai uzuale operații pe care le vor efectua programele va fi repetarea uneia sau mai multor instrucțiuni de un număr dat de ori. Instrucțiunea *for* din C++ permite programelor exact acest lucru. Instrucțiunea *for* folosește o variabilă de control care urmărește numărul de execuții ale buclei. Forma generală a instrucțiunii *for* este următoarea:

```
for (initializare; test; incrementare)
     instructiune;
```

Atunci când începe, bucla *for* atribuie (inițializează) variabilei de control a buclei o valoare de pornire. Apoi, programul testează condiția buclei. Dacă aceasta este adevărată, programul va executa instrucțiunile buclei. În fine, programul incrementează variabila de control a buclei și repetă testarea condiției. Dacă aceasta este adevărată, procesul se repetă. În cazul în care condiția este falsă, bucla *for* se va încheia și programul va continua execuția cu prima instrucțiune care urmează după instrucțiunea *for*.

Iterarea într-o buclă while

Aşa cum tocmai aţi văzut, bucla *for* din C++ permite unui program să repete una sau mai multe instrucţiuni de un număr dat de ori. În unele cazuri, însă, programele trebuie să repete anumite instrucţiuni cât timp o condiţie este adevărată. De exemplu, în lecţii ulterioare veţi învăţa să citiţi conţinutul unui fişier printr-un program C++. Astfel de programe vor continua să itereze până când ating sfârşitul fişierului. Pentru situaţiile în care un program trebuie să itereze atât timp cât o anume condiţie este adevărată, dar nu neapărat de un număr dat de ori, acesta poate folosi instrucţiunea *while* din C++. Forma generală a instrucţiunii *while* este prezentată aici:

```
while (conditie_este_adevarata)
     instructiune;
```

Atunci când întâlneşte o instrucţiune *while*, programul testează mai întâi condiţia specificată. Dacă aceasta este adevărată, programul va executa instrucţiunile buclei *while*. După executarea ultimei instrucţiuni din buclă, *while* testează din nou condiţia. În cazul în care condiţia rămâne adevărată, instrucţiunile buclei vor fi repetate, iar procesul va continua. Când condiţia devine în cele din urmă falsă, bucla se va încheia, iar programul îşi va continua execuţia cu prima instrucţiune care urmează buclei.

Programul următor, *Get_YN.CPP*, vă solicită să tastaţi *Y* pentru da şi *N* pentru nu. Programul foloseşte apoi o buclă *while* pentru a citi caractere de la tastatură până când utilizatorul tastează *Y* sau *N*. Dacă tasta apăsată este alta decât *Y* sau *N*, programul va genera un sunet în difuzorul calculatorului prin trimiterea caracterului '\a' către fluxul de ieşire *cout*:

```
#include <iostream.h>

void main(void)
  {
    int done = 0; // Devine adevarat cand se citeste Y sau N
    char letter;

    while (! done)
    {
      cout << "\nType Y or N and press Enter to continue: ";
      cin >> letter;

      if ((letter == 'Y') || (letter == 'y'))
        done = 1;
      else if ((letter == 'N') || (letter == 'n'))
        done = 1;
```

```
        else
            cout << '\a'; // Genereaza un sunet in difuzor
                          // pentru a indica un caracter
                          // eronat
    }
    cout << "The letter you typed was " << letter << endl;
}
```

După cum puteți vedea, bucla *while* acceptă, de asemenea, mai multe instrucțiuni grupate între acolade. În exemplul nostru, programul folosește variabila *done* pentru a controla bucla. Cât timp programul nu s-a terminat (utilizatorul nu a tastat Y sau N), bucla continuă execuția. Dacă utilizatorul tastează Y sau N, programul fixează variabila *done* la adevărat și bucla se încheie. Odată ce programele dumneavoastră vor începe să lucreze cu fișiere, veți utiliza buclele *while* în mod regulat.

Iterarea până la îndeplinirea unei anumite condiții



Pe măsură ce programele vor spori în complexitate, vor fi situații în care va trebui să executați o serie de instrucțiuni înrudite până la îndeplinirea unei anumite condiții. De exemplu, un program ar putea calcula valorile salariilor pentru toți angajații unei firme. În acest scop, programul ar trebui să itereze până când a prelucrat și ultimul angajat. Pentru a repeta instrucțiuni până la îndeplinirea unei anumite condiții, programele folosesc de regulă instrucțiunea *while*:

```
while (conditie_este_adevarata)
    instructiune;
```

Atunci când întâlnește o instrucțiune *while*, programul evaluează condiția buclei. În cazul în care condiția este adevărată, programul va executa instrucțiunile din bucla *while*. După executarea ultimei instrucțiuni din buclă, programul testează din nou condiția. Dacă aceasta este adevărată, programul va repeta întregul proces, executând instrucțiunile și repetând testarea condiției. Atunci când condiția este falsă, programul își continuă execuția cu prima instrucțiune care urmează instrucțiunii *while*.

Executarea instrucțiunilor cel puțin o dată

Așa cum tocmai ați aflat, bucla *while* din C++ permite programelor repetarea unei serii de instrucțiuni cât timp o condiție anume este adevărată. Atunci când întâlnește o instrucțiune *while*, programul evaluează mai întâi condiția specificată. Dacă aceasta este adevărată, programul va executa bucla. În cazul în care condiția este falsă, programul nu va executa niciodată instrucțiunile buclei. În funcție de scopul programului, vor fi multe situații în care veți dori să executați o serie de instrucțiuni cel puțin o dată, iar după aceea, pe baza unei

condiții, să repetați eventual execuția. În astfel de cazuri, programele pot utiliza o buclă *do while*, așa cum este ilustrat aici:

```
do {
     instructiuni;
} while (conditie_este_adevarata);
```

Atunci când întâlnește o buclă *do while*, programul intră în buclă și începe execuția instrucțiunilor conținute. După executarea ultimei instrucțiuni din buclă, programul evaluează condiția specificată. Dacă aceasta este adevărată, programul va reveni cu execuția la începutul buclei, așa cum se vede aici:

```
do { <-----------------------------------+
     instructiuni;                       |
} while (conditie_este_adevarata); ------+
```

În cazul în care condiția este falsă, programul nu va repeta instrucțiunile din buclă, ci în schimb își va continua execuția cu prima instrucțiune care urmează buclei. O folosire uzuală a buclei *do while* este pentru afișarea opțiunilor de meniu și prelucrarea ulterioară a selecției utilizatorului. Veți dori ca programul să afișeze meniul cel puțin o dată. Dacă utilizatorul selectează orice altă opțiune decât Quit, programul va îndeplini comanda aleasă și apoi va afișa din nou meniul (repetând instrucțiunile din buclă). Dacă utilizatorul selectează Quit, bucla se va încheia și programul își va continua operațiile cu prima instrucțiune de după buclă.

Repetarea instrucțiunilor cât timp o condiție este adevărată



În funcție de nevoile unui program, ar putea fi situații în care acesta va trebui să execute o serie de instrucțiuni cel puțin o dată și apoi să repete, eventual, instrucțiunile respective dacă o anumită condiție este adevărată. În astfel de cazuri, programele ar trebui să folosească instrucțiunea *do while* din C++, așa cum este prezentată aici:

```
do {
     instructiune;
} while (conditie_este_adevarata);
```

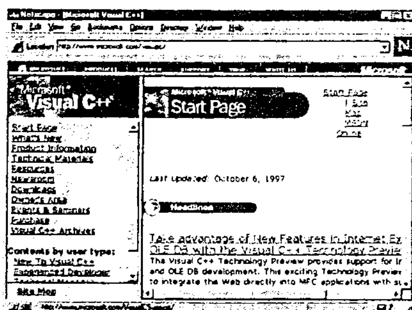
Atunci când întâlnește o instrucțiune *do while*, un program execută imediat instrucțiunile conținute în buclă. În continuare, programul examinează condiția buclei. Dacă aceasta este adevărată, programul va repeta instrucțiunile buclei și procesul va continua. Atunci când condiția buclei devine falsă, programul își continuă execuția cu prima instrucțiune ce urmează instrucțiunii *do while*.

Ce trebuie să știți

Prelucrarea iterativă este capacitatea unui program de a repeta una sau mai multe instrucțiuni. Lecția de față v-a prezentat instrucțiunile iterative (sau de buclă) din C++. Așa cum ați văzut, instrucțiunea *for* permite programelor să repete una sau mai multe instrucțiuni de un număr dat de ori. Instrucțiunea *while* permite programelor să repete o serie de instrucțiuni atât timp cât o condiție anume este adevărată. În fine, instrucțiunea *do while* permite programelor să execute o serie de instrucțiuni cel puțin o dată, repetând-o eventual în cazul în care o condiție anume este adevărată. În lecția 10, „O introducere în funcții", veți învăța să împărțiți programele mari în fragmente mai mici și mai ușor de gestionat, numite funcții. Dar înainte de a trece la lecția 10, asigurați-vă că ați reținut următoarele aspecte importante:

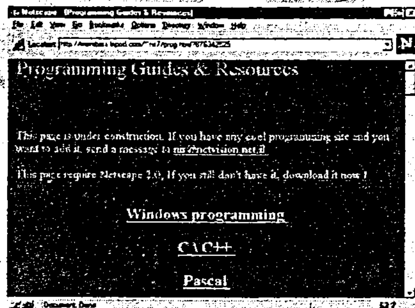
- ☑ Instrucțiunea *for* din C++ permite programelor să repete una sau mai multe instrucțiuni de un număr dat de ori.
- ☑ Instrucțiunea *for* este alcătuită din patru părți: o inițializare, o condiție de test, instrucțiunile de repetat și o incrementare.
- ☑ Instrucțiunea *for* nu impune programelor incrementarea variabilei de control a buclei cu unu și nu obligă nici la o numărare crescătoare.
- ☑ Bucla *while* din C++ permite programelor să repete instrucțiuni atât timp cât o anumită condiție este adevărată.
- ☑ Programele folosesc frecvent bucla *while* pentru a citi conținutul unui fișier, iterând până când programul ajunge la sfârșitul fișierului.
- ☑ Instrucțiunea *do while* din C++ permite programelor să execute una sau mai multe instrucțiuni cel puțin o dată, repetându-le eventual în funcție de o condiție anume.
- ☑ Programele folosesc adesea *do while* pentru operații cu meniuri.
- ☑ Atunci când condiția dintr-o buclă *for*, *while* sau *do while* devine falsă, programul își va continua execuția cu prima instrucțiune care urmează buclei.

MICROSOFT VISUAL C++



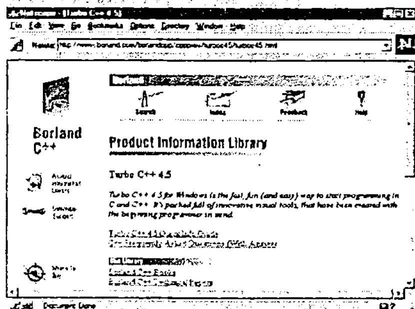
http://www.microsoft.com/visualc/

RESURSE ŞI INSTRUCŢIUNI DE PROGRAMARE



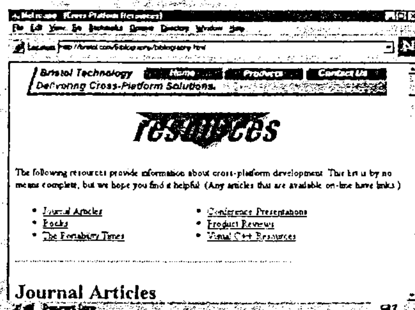
http://members.tripod.com/~ntr7/prog.html?876342525

TURBO C++ 4.5



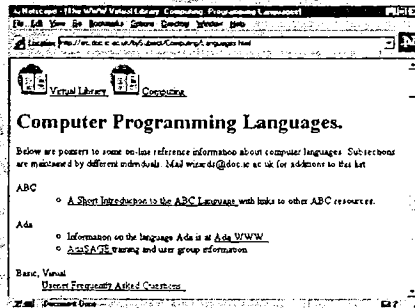
http://www.borland.com/borlandcpp/cppprev/-turboc45/turboc45.html

RESURSE PENTRU DIVERSE PLATFORME



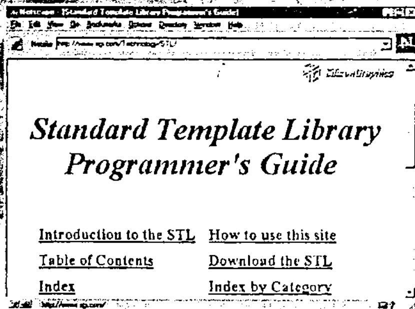
http://bristol.com/Bibliography/bibliography.html

BIBLIOTECA VIRTUALĂ ÎN WWW



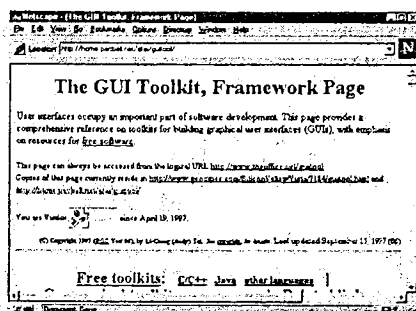
http://src.doc.ic.ac.uk/bySubject/Computing/Languages.html

BIBLIOTECA DE ŞABLOANE STANDARD



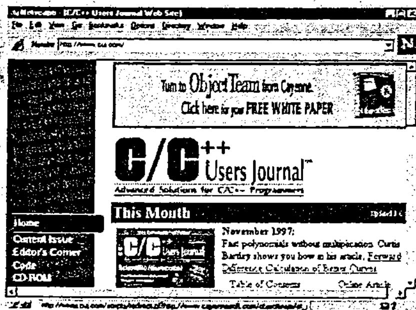
http://www.sgi.com/Technology/STL/

KITUL CU INSTRUMENTE GUI



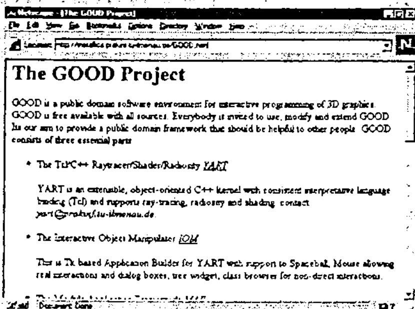
http://home.pacbell.net/atai/guitool/

REVISTA PROGRAMATORILOR ÎN C++



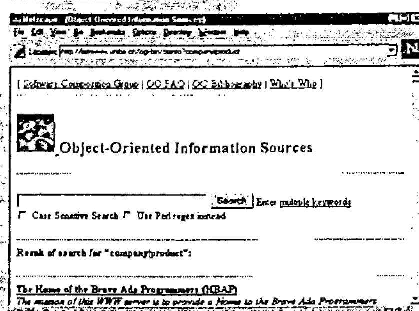
http://www.cuj.com/

PROIECTUL GOOD



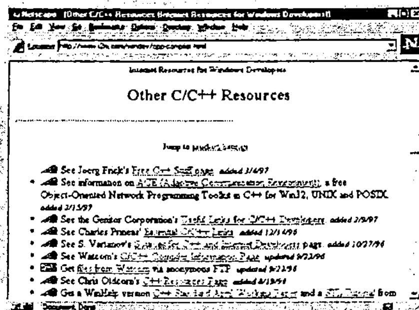
http://metallica.prakinf.tu-ilmenau.de/ GOOD.html

INFORMAȚII DESPRE ORIENTAREA SPRE OBIECT



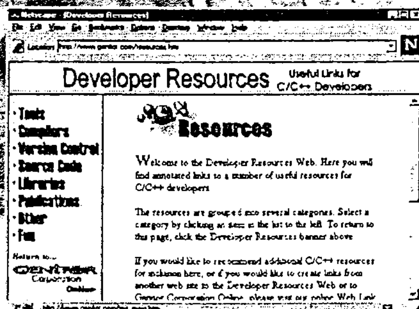
http://iamwww.unibe.ch/cgi-bin/ ooinfo?company/product

ALTE RESURSE C/C++



http://www.r2m.com/windev/ cpp-compiler.html

RESURSE PENTRU PROGRAMATORI



http://www.genitor.com/ resources.htm

CELE MAI FRECVENTE ÎNTREBĂRI

G++ FAQ

Frequently asked questions about the GNU C++ compiler

September 14, 1997

http://www.cygnus.com/misc/g++FAQ_toc.html

APELURI INVERSE ÎN C++

Callbacks in C++

Free C++ Callback Library - Callback 1.0

http://www.primenet.com/~jakubik/callback.html

FIŞIER HEADER PENTRU CLASA ARRAY

Header File for an Array Class

Note:

Table of Contents

http://www.informatik.uni-konstanz.de/~kuebl/c++/array.h.html

BAZE DE DATE ALE RESURSELOR DE PROGRAMARE

DOPER

http://www-ug.eecg.toronto.edu/~sokorac/Doper/VisualCPP.html

CLASELE C++

C++ classes

http://www.ts.umu.se/~maxell/C++/

RESURSE VISUAL C++

AUL Visual C++

http://www.aul.llu.edu/tech/visualc.html

PARTEA a II-a

Crearea programelor cu ajutorul funcțiilor

Pe măsură ce programele dumneavoastră vor spori în dimensiuni și în complexitate, veți decide să le împărțiți în fragmente mai mici și mai ușor de gestionat, numite *funcții*. Fiecare funcție dintr-un program va îndeplini o sarcină bine definită. Să presupunem, de pildă, că scrieți un program de contabilitate. Ați putea avea atunci o funcție pentru gestionarea creditelor, o alta pentru debite, încă o funcție pentru salarii și tot așa. Concentrându-vă asupra unei singure funcții la un moment dat, programele vor deveni mai ușor de creat și de înțeles. În plus, veți descoperi că multe funcții create într-un program pot fi utilizate într-un altul, ceea ce vă economisește timpul destinat programării. Lecțiile conținute în această parte sunt următoarele:

Lecția 10

O introducere în funcții

Pe măsură ce programele dumneavoastră cresc în dimensiune și în complexitate, ar trebui să le împărțiți în fragmente mai mici și mai ușor de gestionat, numite *funcții*. Fiecare funcție din program trebuie să execute o anumită sarcină. De exemplu, dacă ați scrie un program de gestiune a salariilor, ați putea crea o funcție care determină numărul de ore lucrate de un angajat, o altă funcție care calculează indemnizația pentru orele suplimentare, o a treia funcție care tipărește chitanțele de salariu și așa mai departe. Atunci când ar trebui să efectueze o anumită operație, programul *apelează* funcția corespunzătoare, transmițând funcției informațiile care îi sunt necesare pentru a-și îndeplini sarcina, cum ar fi numele angajatului sau salariul pe oră. Lecția de față vă va învăța să creați și să folosiți funcții în cadrul programelor C++. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- *Funcțiile* grupează instrucțiuni înrudite pentru a îndeplini o anumită sarcină.
- Pentru a folosi o funcție, un program *apelează* funcția respectivă specificând numele funcției, urmat de paranteze, ca de pildă *beep()*.
- După ce efectuează operațiile corespunzătoare, multe funcții întorc o valoare de un tip anume, precum *int* sau *float*, valoare pe care programul o poate testa sau o poate atribui unei variabile.
- Programele pot transmite parametri (informații) către funcții, așa cum ar fi numele, vârsta sau salariul unui angajat, specificând respectivii parametri între parantezele care urmează numelui de funcție.
- C++ folosește *prototipuri* de funcție pentru a defini tipul valorii pe care o funcție o întoarce programului, precum și numărul și tipul parametrilor pe care programul îi transmite funcției.

Odată ce programele vor fi tot mai mari și mai puternice, utilizarea funcțiilor va deveni esențială. Așa cum veți vedea, crearea și utilizarea funcțiilor în C++ este foarte facilă.

Crearea și utilizarea primelor dumneavoastră funcții

La crearea programelor ar trebui să gândiți fiecare funcție astfel încât să efectueze o sarcină bine definită. Dacă descoperiți că o funcție îndeplinește mai multe acțiuni, ar trebui să împărțiți acea funcție în două sau mai multe funcții. Este bine să atribuiți un nume unic fiecărei funcții pe care o creați într-un program. Ca și în cazul numelor de variabile, numele pe care le alegeți pentru funcții ar trebui să corespundă operațiilor pe care acestea le efectuează. De exemplu, prin simpla parcurgere a numelor de funcții din tabelul 10 puteți să vă faceți deja o idee despre scopul fiecărei funcții.

Numele funcției	Scopul funcției
tipareste_note_test	Tipărește notele de la un test
debite	Gestionează debitele unei firme
citeste_nume_utilizator	Solicită utilizatorului să introducă numele său
tipareste_document	Tipărește documentul specificat
calcul_impozit_venit	Calculează impozitul pe venit al unui utilizator

Tabelul 10 *Exemple de nume de funcții sugestive.*

O funcție C++ este asemănătoare ca structură cu programul principal (*main*) pe care l-ați folosit în toate programele de până acum. Cu alte cuvinte, numele funcției este precedat de un tip, precum *int* sau *void*, și urmat de o listă de parametri care apar între paranteze. De asemenea, ca și în cazul funcției *main*, instrucțiunile funcției trebuie grupate cu ajutorul acoladelor, așa cum se vede aici:

```
tip_intors nume_functie(lista_parametri)
  {
       declaratii_variabile;
       instructiuni;
  }
```

Să vedem, spre exemplu, cum corespunde structura unei funcții cu cea a programului principal (*main*):

```
tip nume(lista_parametri)        void main(void)
  {                                {
    declaratii_variabile; |------| int count;
    instructiuni; |-----------------| for (count = 0;
                                    |   count < 10; count++)
                                    |     cout << count << ' ';
  }                                }
```

Instrucțiunile următoare definesc o funcție numită *show_message* care afișează pe ecran un mesaj prin intermediul *cout*:

```
void show_message(void)
   {
     cout << "Hello, I've been Rescued by C++" << endl;
   }
```

După cum probabil vă amintiți din lecția 3, „O privire mai atentă asupra limbajului C++", cuvântul *void* care precede numelui de funcție arată (compilatorului de C++ și programatorilor care citesc codul) că funcția nu întoarce nici o valoare către apelantul său. Similar, cuvântul *void* aflat între paranteze specifică faptul că funcția nu folosește *parametri* (informații pe care programul le transmite unei funcții). Programul următor, *Show_Msg.CPP*, folosește funcția *show_message* pentru a afișa pe ecran un mesaj:

```
#include <iostream.h>

void show_message(void)
   {
     cout << "Hello, I've been Rescued by C++" << endl;
   }

void main(void)
   {
     cout << "About to call the function" << endl;
     show_message();
     cout << "Back from the function" << endl;
   }
```

După cum ați învățat, execuția unui program începe întotdeauna cu *main*. În exemplul nostru, cea de a doua instrucțiune din *main*, *apelul de funcție*, invocă funcția *show_message*:

```
show_message();
```

Parantezele care urmează după numele funcției arată compilatorului de C++ că programul utilizează o funcție. Ulterior, în această lecție, veți vedea că programele au posibilitatea de a transmite funcțiilor informații (parametri) tocmai între aceste paranteze. Atunci când compilați și rulați programul *Show_Msg.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> Show_Msg <Enter>
About to call the function
Hello, I've been Rescued by C++
Back from the function
```

La întâlnirea unui apel de funcție, programul începe să execute instrucțiunile aflate în interiorul funcției. După executarea tuturor instrucțiunilor unei funcții (cu alte cuvinte, după ce funcția își încheie sarcina), execuția programului continuă cu instrucțiunea imediat următoare apelului de funcție, așa cum se vede în continuare:

```
#include <iostream.h>

void show message(void)
  {
    cout << "Hello, I've been Rescued by C++" << endl;
  }

void main(void)
  {
    cout << "About to call the function" << endl;
    show message();
    cout << "Back from the function" << endl;
  }
```

În exemplul nostru, programul execută prima instrucțiune din *main*, care afișează un mesaj ce informează utilizatorul că programul este pe punctul de a apela funcția. Apoi, programul întâlnește apelul de funcție și începe să execute instrucțiunile aflate în funcția *show_message*. După executarea unicei instrucțiuni a funcției, programul revine în funcția *main*, la instrucțiunea imediat următoare apelului de funcție. În acest moment, programul afișează un mesaj care informează utilizatorul că apelul de funcție s-a încheiat și apoi își încheie execuția. Programul următor, *Two_Msgs.CPP*, utilizează două funcții, *show_title* și *show_lessons*, pentru a afișa informații despre această carte:

```
#include <iostream.h>

void show_title(void)
  {
    cout << "Book: Rescued by C++" << endl;
  }

void show_lesson(void)
  {
    cout << "Lesson: Getting Started With Functions"
      << endl;
  }
```

```
void main(void)
  {
    show_title();
    show_lesson();
  }
```

Atunci când își începe execuția, programul va apela mai întâi funcția *show_title*, care afișează un mesaj prin intermediul *cout*. După ce *show_title* se încheie, programul apelează funcția *show_lesson*, care afișează la rândul ei un mesaj. După ce se încheie și *show_lesson*, în funcția *main* nu a mai rămas nici o instrucțiune de executat, așa că programul se termină.

Funcțiile din programele prezentate până acum în această lecție au îndeplinit sarcini foarte simple. În fiecare caz, programul ar fi putut foarte bine să efectueze aceleași operații fără a mai recurge la funcții, incluzând pur și simplu instrucțiunile funcțiilor în interiorul programului principal. Scopul acelor funcții a fost, însă, să vă arate modul în care un program definește și ulterior apelează o funcție. Pe măsură ce programele vor spori în complexitate, veți folosi funcții pentru a simplifica operații întinse prin împărțirea acestora în fragmente mai mici și mai ușor de gestionat.

La crearea de funcții veți vedea că, deoarece conțin mai puține linii de cod decât un singur program mare, ele sunt mai ușor de înțeles și mai simplu de modificat. În plus, veți vedea că în multe cazuri puteți să preluați o funcție creată într-un program și să o utilizați neschimbată într-un alt program. Prin crearea unei biblioteci de funcții veți reduce cantitatea de timp pe care ați petrece-o mai târziu scriind și testând funcții similare.

Apelarea unei funcții



O funcție este o mulțime de instrucțiuni înrudite care îndeplinește o anumită sarcină. Prin crearea de funcții în cadrul programelor, operațiile complexe pot fi împărțite în fragmente mai mici și mai ușor de gestionat. Programele execută instrucțiunile unei funcții apelând acea funcție. Pentru a apela o funcție, un program trebuie pur și simplu să specifice numele funcției, urmat de paranteze, ca mai jos:

```
nume_functie();
```

Atunci când un program transmite informații (parametri) către o funcție, acestea vor fi plasate între paranteze, separate prin virgule, așa cum se vede aici:

```
salariu(nume_angajat, id_angajat, salariu);
```

După ce se încheie și ultima instrucțiune din funcție, execuția programului continuă cu prima instrucțiune care urmează apelului de funcție.

Programele pot transmite informații către funcții

Pentru a spori capacitatea unei funcții, C++ permite programelor să transmită către funcții informații (parametri). Atunci când o funcție folosește parametri, va trebui să indicați tipul fiecărui parametru, fie acesta *int*, *float*, *char* și așa mai departe. De exemplu, funcția următoare, *show_number*, folosește un parametru de tip *int*:

```
void show_number(int value)
  {
    cout << "The parameter's value is " << value << endl;
  }
```

Atunci când cheamă funcția *show_number*, programul trebuie să transmită funcției o valoare, ca mai jos:

```
show_number(1001);            Valoare transmisă funcției
```

În culise, C++ va înlocui fiecare apariție din funcție a numelui parametrului cu valoarea pe care programul a transmis-o funcției, așa cum se vede aici:

```
show number(1001);

   void show number(int value)
   {
        cout << "The parameter's value is " << value << endl;
   }

          void show number(1001)
          {
             cout << "The parameter's value is " << 1001
               << endl;
          }
```

După cum puteți vedea, deoarece C++ înlocuiește numele parametrului cu valoarea transmisă, funcția *show_number* afișează valoarea 1001 pe care programul principal a transmis-o prin intermediul apelului de funcție.

Programul următor, *UseParam.CPP*, folosește funcția *show_number* de mai multe ori, transmițând de fiecare dată o altă valoare:

```
#include <iostream.h>

void show_number(int value)
  {
    cout << "The parameter's value is " << value << endl;
  }

void main(void)
  {
    show_number(1);
    show_number(1001);
    show_number(-532);
  }
```

După compilarea şi rularea programului *UseParams.CPP*, pe ecran vor fi afişate următoarele:

```
C:\> UseParam <Enter>
The parameter's value is 1
The parameter's value is 1001
The parameter's value is -532
```

Precum vedeţi, de fiecare dată când programul apelează funcţia, C++ atribuie variabilei *value* valoarea corectă. Experimentaţi puţin cu acest program, modificând valorile pe care programul principal le transmite funcţiei şi remarcaţi rezultatul.

Aşa cum aţi învăţat, pentru ca un program să poată transmite informaţii unei funcţii, aceasta trebuie să accepte unul sau mai mulţi parametri. Fiecare parametru al unei funcţii are un tip propriu. În cazul funcţiei *show_number*, valoarea parametrului trebuie să aibă tipul *int*. Dacă încercaţi să transmiteţi funcţiei o valoare de un alt tip, aşa cum ar fi o valoare în virgulă mobilă, compilatorul va genera o eroare de sintaxă. De cele mai multe ori, programele vor transmite unei funcţii mai multe valori. Pentru fiecare parametru, funcţia trebuie să precizeze un nume şi un tip corespunzător. Spre exemplu, programul următor, *BigSmall.CPP*, foloseşte funcţia *show_big_and_little* pentru a afişa cel mai mare şi cel mai mic dintre cele trei numere întregi pe care le primeşte:

```
#include <iostream.h>

void show big_and_little(int a, int b, int c)
  {
    int small = a;
    int big = a;
```

```
      if (b > big)
         big = b;
      if (b < small)
         small = b;
      if (c > big)
         big = c;
      if (c < small)
         small = c;

      cout << "The biggest value is  << big << endl;
      cout << "The smallest value is  << small << endl;
   }

void main(void)
   {
      show_big_and_little(1, 2, 3);
      show_big_and_little(500, 0, -500);
      show_big_and_little(1001, 1001, 1001);
   }
```

La apelarea funcției de către program, C++ fixează valorile parametrilor ca mai jos:

```
show_big_and_little(1, 2, 3);

      void show_big_and_little(int a, int b, int c);
```

După compilarea și rularea programului *BigSmall.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> BigSmall <Enter>
The biggest value is 3
The smallest value is 1
The biggest value is 500
The smallest value is -500
The biggest value is 1001
The smallest value is 1001
```

În cadrul exemplului următor, programul *Show_Emp.CPP* utilizează funcția *show_employee* pentru a afișa vârsta (de tip *int*) și salariul (de tip *float*) unui angajat:

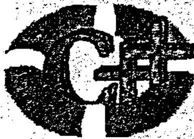
```
#include <iostream.h>

void show_employee(int age, float salary)
  {
    cout << "The employee is " << age << " years old"
      << endl;
    cout << "The employee makes $" << salary << endl;
  }

void main(void)
  {
    show_employee(32, 25000.00);
  }
```

După cum puteţi vedea, funcţia *show_employee* defineşte parametrii de tip *int* şi *float*, aşa cum este necesar.

Transmiterea de parametri către funcţii



Atunci când o funcţie foloseşte parametri, ea trebuie să precizeze pentru fiecare parametru un nume unic şi un tip. La apelarea funcţiei de către program, C++ va atribui valorile de parametri numelor de parametri din funcţie, de la stânga la dreapta. Fiecare parametru al unei funcţii are un tip anume, precum *int*, *float* sau *char*. Valorile pe care programul le transmite unei funcţii prin intermediul parametrilor trebuie să fie conforme cu tipurile respectivilor parametri.

Funcţiile pot întoarce un rezultat apelantului

Aşa cum am discutat, funcţiile unui program ar trebui să îndeplinească sarcini precise. De multe ori, funcţiile efectuează un anumit calcul. În acest caz, funcţiile întorc apelantului (partea din program care a apelat funcţia) un rezultat. Atunci când o funcţie întoarce o valoare este necesar să indicaţi tipul acelei valori, cum ar fi *int*, *float*, *char* şi aşa mai departe.

Pentru a informa C++ cu privire la tipul întors de o funcţie, este suficient să precizaţi tipul în cauză înaintea numelui funcţiei. Spre exemplu, funcţia următoare, *add_values*, adună doi parametri întregi şi întoarce un rezultat de tipul *int* către programul apelant:

```
int add_values(int a, int b)
  {
  int result;
  result = a + b;
  return(result);
  }
```

În acest caz, cuvântul *int* care apare înainte de numele funcției precizează *tipul întors* de funcție. Pentru a întoarce o valoare către apelant, funcțiile trebuie să folosească instrucțiunea *return*. Atunci când întâlnește instrucțiunea *return*, o funcție întoarce către apelant valoarea indicată și își încheie execuția, cedând controlul apelantului. Instrucțiunea următoare, de pildă, atribuie variabilei *result* valoarea întoarsă de funcția *add_values*:

```
result = add_values(1, 2);
```

În acest exemplu, programul atribuie valoarea întoarsă de funcție unei variabile. Programul poate de asemenea să afișeze direct valoarea întoarsă de o funcție prin intermediul *cout*, ca mai jos:

```
cout << "Sum of values is " << add_values(500, 501) << endl;
```

Implementarea anterioară a funcției *add_values* folosea trei instrucțiuni pentru a face mai ușor de înțeles modul de operare a funcției. Puteți, însă, reduce funcția la o singură instrucțiune *return*, așa cum se ilustrează în continuare:

```
int add_values(int a, int b)
  {
    return(a+b);
  }
```

Programul următor, *AddValue.CPP*, utilizează funcția *add_values* pentru a aduna diferite numere:

```
#include <iostream.h>

int add_values(int a, int b)
  {
    return(a+b);
  }
```

```
void main(void)
   {
     cout << "100 + 200 = " << add_values(100, 200) << endl;
     cout << "500 + 501 = " << add_values(500, 501) << endl;
     cout << "-1 + 1 = " << add_values(-1, 1) << endl;
   }
```

Experimentați puțin programul *AddValue.CPP*, modificând valorile pe care programul le transmite funcției. Puteți încerca să transmiteți funcției valori mari, cum ar fi 20000 și 30000. După cum intuiți probabil, funcția (care întoarce o valoare de tip *int*) va produce o eroare de depășire și va întoarce un rezultat eronat.

Nu toate funcțiile întorc o valoare de tip *int*. Funcția următoare, *average_value*, întoarce media a două numere întregi, aceasta putând fi o valoare zecimală, precum 3,5:

```
float average_value(int a, int b)
   {
     return((a + b) / 2.0);
   }
```

În acest caz, cuvântul *float* care precede numele funcției specifică tipul valorii întoarse de funcție. Programul următor, *GetAve.CPP*, utilizează funcția *average_value* pentru a afișa media numerelor 5 și 10:

```
#include <iostream.h>

float average_value(int a, int b)
   {
     return((a + b) / 2.0);
   }

void main(void)
   {
     cout << "The average value is: " << average_value(5, 10)
       << endl;
   }
```

Funcții care nu întorc valori

Așa cum veți vedea, nu toate funcțiile întorc un rezultat. Spre exemplu, anterior în această lecție ați folosit funcția *show_message* pentru a afișa un mesaj prin intermediul *cout*. Dacă o funcție nu întoarce nici un rezultat, numele funcției trebuie precedat de tipul *void*. După cum ați aflat, pentru a întoarce o valoare către apelant, funcțiile folosesc instrucțiunea *return*. Atunci când întâlnește o instrucțiune *return*, o funcție își încheie execuția, iar C++ întoarce apelantului valoarea specificată. Inspectând programe C++, ați putea întâlni situații în care veți găsi în cadrul unei funcții o instrucțiune *return* care nu întoarce nici o valoare, ca mai jos:

```
return;
```

În acest exemplu, tipul funcției este *void* (nu întoarce nici o valoare), iar instrucțiunea *return* încheie pur și simplu execuția funcției.

Notă: *Dacă funcția conține instrucțiuni ce apar după instrucțiunea **return**, acestea nu vor mai fi executate. Așa cum am discutat, atunci când întâlnește o instrucțiune **return**, o funcție întoarce către apelantul său valoarea specificată, iar execuția programului continuă cu prima instrucțiune care urmează apelului de funcție.*

Utilizarea valorii întoarse de o funcție

Atunci când o funcție întoarce o valoare, apelantul poate atribui acea valoare unei variabile prin intermediul operatorului de atribuire, așa cum se vede aici:

```
suma_salariu = salariu(angajat, ore, salariu_ora);
```

În plus, apelantul poate pur și simplu să specifice numele funcției. Spre exemplu, instrucțiunea următoare afișează prin intermediul *cout* valoarea întoarsă de o funcție:

```
cout << "Angajatul este platit cu " << salariu(angajat, ore,
  salariu_ora) << endl;
```

De asemenea, apelantul poate folosi valoarea întoarsă de o funcție în cadrul unei condiții, ca în continuare:

```
if (salariu(angajat, ore, salariu_ora) << 500.00)
   cout << "Acest angajat are nevoie de o marire de salariu"
     << endl;
```

Precum vedeți, valoarea întoarsă de o funcție poate fi folosită de program în diverse feluri.

Despre prototipurile de funcții

Înainte ca un program să poată apela o funcție, C++ trebuie să cunoască tipul valorii pe care o întoarce funcția și numărul și tipul parametrilor utilizați de funcție. În fiecare dintre programele acestei lecții, definiția funcției apelate de un program se afla în fișierul sursă înaintea apelului de funcție. De multe ori, însă, funcțiile apar în diferite părți ale fișierului sursă și se întâmplă adesea ca o funcție să apeleze o alta.

Pentru a vă asigura că C++ cunoaște detaliile corespunzătoare fiecărei funcții utilizate în program, ar trebui ca spre începutul fișierului sursă să plasați *prototipurile funcțiilor*. În general, prototipul unei funcții informează C++ (ca și un alt programator care vă citește programul) despre tipul întors de o funcție și despre parametrii acesteia. Instrucțiunile următoare prezintă prototipuri de funcții pentru mai multe dintre funcțiile folosite pe parcursul acestei lecții:

```
void show_message(void);

void show_number(int);

void show_employee(int, float);

int add_value(int, int);

float average_value(int, int);
```

După cum puteți vedea, prototipul unei funcții indică tipul valorii întoarse de funcție și numărul și tipul parametrilor. Remarcați semnul punct și virgulă care încheie fiecare prototip:

```
float average value(int, int);
```

Tipul valorii întoarse

Tipurile parametrilor

Dacă un program încearcă apelarea unei funcții pentru care C++ nu a întâlnit încă definiția sau un prototip, compilatorul va genera o eroare de sintaxă. Pe măsură ce veți parcurge fișiere de antet din C++ sau alte programe, veți întâlni prototipuri de funcții în mod regulat. Programul următor, *Proto.CPP*, ilustrează modul de utilizare al prototipului unei funcții:

```
#include <<iostream.h>

float average_value(int, int); // Prototipul functiei

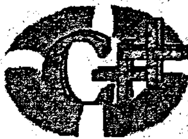
void main(void)
   {
```

```
    cout << "The average of 2000 and 2 is " <<
      average_value(2000, 2) << endl;
  }

float average_value(int a, int b)
  {
    return((a + b) / 2.0);
  }
```

În acest caz, programul apelează funcția *average_value* înaintea definiției acesteia. Din această cauză, programul utilizează prototipul funcției, care precede definiția funcției *main*. Dacă ștergeți prototipul funcției și compilați apoi programul *Proto.CPP*, compilatorul de C++ va genera erori de sintaxă.

Prototipurile de funcții vă vin în ajutor



Prototipul unei funcții precizează compilatorului de C++ tipul valorii întoarse de o funcție și numărul și tipul parametrilor acelei funcții. La compilarea unui program, compilatorul utilizează fiecare prototip de funcție pentru a se asigura că nu folosiți eronat valoarea întoarsă de vreo funcție (atribuind, de pildă, o valoare întoarsă de tip *float* unei variabile de tip *int*) și că nu transmiteți o valoare nepotrivită pentru vreun parametru. În trecut, multe dintre compilatoarele de C nu efectuau asemenea verificări de tip. Ca o consecință, programatorii petreceau adesea ore întregi încercând să depaneze erori care apăruseră pentru că unei funcții care aștepta o valoare de tip *float* îi era transmisă o valoare de tip *int*. Atunci când vi se semnalează o eroare datorată unui conflict cu un prototip de funcție, fiți recunoscător. În trecut, compilatorul nu ar fi sesizat eroarea și programul dumneavoastră pur și simplu nu ar fi funcționat.

Ce trebuie să știți

În lecția de față ați învățat să folosiți funcții în cadrul programelor C++. Această lecție s-a oprit asupra multor aspecte importante, cum ar fi parametrii, tipurile valorilor întoarse și prototipurile de funcții. Ar putea fi o idee bună să mai experimentați câteva minute cu exemplele de programe din această lecție. În lecția 11, „Modificarea valorilor parametrilor", veți învăța să modificați valorile parametrilor în cadrul funcțiilor. Dar înainte de a trece la lecția 11, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Pe măsură ce programele dumneavoastră cresc în dimensiune și în complexitate, ar trebui să împărțiți aceste programe în fragmente mai mici și mai ușor de gestionat, numite funcții. Fiecare funcție trebuie să aibă un nume unic. Alegeți nume de funcții care să descrie suficient de bine operațiile efectuate de funcții.

- ☑ Funcţiile pot întoarce valori către apelant. Dacă o funcţie întoarce o valoare, va trebui să specificaţi tipul acestei valori (*int*, *char* şi aşa mai departe) înaintea numelui de funcţie; în caz contrar, ar trebui să precizaţi *void* înainte de numele funcţiei.

- ☑ Programele transmit informaţii către funcţii prin intermediul parametrilor. Dacă o funcţie primeşte parametri, va trebui să specificaţi pentru fiecare parametru un nume unic şi un tip. Dacă o funcţie nu primeşte parametri, ar trebui să plasaţi între parantezele care urmează numelui de funcţie cuvântul cheie *void*.

- ☑ C++ trebuie să cunoască tipul valorii întoarse de o funcţie şi numărul şi tipul parametrilor pe care îi primeşte funcţia. Dacă definiţia unei funcţii se află după apelul acesteia, atunci va trebui să plasaţi la începutul fişierului sursă prototipul funcţiei respective.

Lecția 11
Modificarea valorilor parametrilor

În lecția 10, „O introducere în funcții", ați învățat să împărțiți programele în fragmente mai mici și mai ușor de gestionat, numite funcții. Așa cum ați văzut, programele au posibilitatea de a transmite informații (parametri) către funcții. Funcțiile din cadrul lecției 10 foloseau sau afișau valorile parametrilor, dar nu le modificau. În lecția de față veți învăța să modificați valorile parametrilor în cadrul funcțiilor. După cum veți afla, modificarea unui parametru în interiorul unei funcții necesită mai mulți pași decât ați crede. Această lecție vă va arăta, însă, toți pașii care sunt necesari. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- O funcție nu poate modifica valoarea unui parametru decât în cazul în care folosește pointeri sau referințe C++.
- Pentru a modifica valoarea unui parametru, o funcție trebuie să cunoască adresa din memorie a acelui parametru.
- Operatorul de adresare (&) din C++ permite programelor să determine adresa de memorie a unei variabile.
- Atunci când cunosc o adresă de memorie, programele pot folosi operatorul de indirectare a memoriei (*) din C++ pentru a determina valoarea plasată la acea adresă.
- Atunci când o funcție trebuie să modifice valoarea unui parametru, programul trebuie să transmită funcției adresa acelui parametru.

Așa cum veți vedea, modificarea valorii unui parametru în cadrul unei funcții este o operație uzuală. Experimentați cu programele prezentate în această lecție pentru a vă familiariza cât mai bine cu această operație.

De ce nu pot în mod normal funcțiile să schimbe valorile parametrilor

Programul următor, *NoChange.CPP*, transmite funcției *display_values* doi parametri, numiți *big* și *small*. Funcția *display_values* atribuie ambilor parametri valoarea 1001 și apoi afișează valorile acestora. După ieșirea din funcție, programul afișează la rândul său valorile celor doi parametri:

```
#include <iostream.h>

void display_values(int a, int b)
  {
    a = 1001;
    b = 1001;

    cout << "The values within display values are " << a <<
        and " << b << endl;
  }

void main(void)
  {
    int big = 2002, small = 0;

    cout << "Values before function " << big <<  and " <<
      small << endl;

    display_values(big, small);

    cout << "Values after function  << big << " and " <<
       small << endl;
  }
```

La compilarea și rularea programului *NoChange.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> NoChange <Enter>
Values before function 2002 and 0
The values within display_values are 1001 and 1001
Values after function 2002 and 0
```

După cum puteți vedea, funcția *display_values* modifică valoarea fiecărei parametru la 1001. Cu toate acestea, după încheierea funcției valorile variabilelor *big* și *small* sunt neschimbate. Pentru a înțelege de ce atribuirile efectuate în funcție nu au afectat variabilele *big* și *small* din programul principal va trebui să înțelegeți modul în care C++ transmite parametrii către funcții.

În mod implicit, atunci când un program transmite un parametru unei funcții, C++ face o copie a valorii parametrului și depune acea copie într-o locație temporară de memorie care se numește *stivă*. Funcția utilizează apoi *copia* valorii pentru efectuarea diverselor operații. La încheierea funcției, C++ ignoră conținutul stivei și orice modificări pe care funcția le-a operat asupra copiilor valorilor de parametri.

Așa cum știți, o variabilă este un nume pe care programul îl asociază cu o locație de memorie ce conține o valoare de un anumit tip. Să presupunem, de exemplu, că variabilele *big* și *small* s-ar afla în locațiile de memorie 10 și 12. Atunci când transmiteți

aceste variabile funcției *display_values*, C++ plasează pe stivă copii ale valorilor variabilelor. Așa cum o înfățișează figura 11.1, funcția *display_values* va utiliza copiile valorilor de variabile.

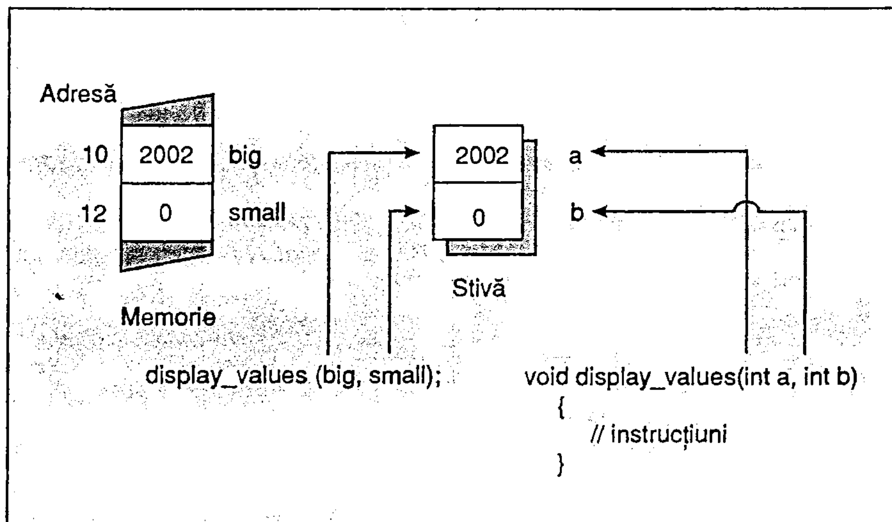


Figura 11.1 *C++ plasează copii ale valorilor parametrilor într-o locație temporară numită stivă.*

După cum puteți vedea, funcția *display_value* are acces la conținutul stivei care conține copiile valorilor 2002 și 0. Deoarece funcția *display_value* nu cunoaște locațiile de memorie (10 și 12) ale variabilelor *big* și *small*, ea nu are cum să modifice valorile propriu-zise ale acestor variabile.

De ce funcțiile C++ nu pot în mod normal să schimbe valorile parametrilor



Atunci când un program transmite parametri unei funcții, C++ plasează copii ale valorilor respectivilor parametri într-o locație de memorie temporară numită stivă. Orice modificări pe care funcția le efectuează asupra parametrilor afectează numai copiile de pe stivă ale variabilelor. La încheierea funcției, C++ distruge conținutul stivei, ignorând orice modificări pe care funcția le-a adus valorilor parametrilor. Cum funcția nu cunoaște nimic despre adresele de memorie ale parametrilor, ea nu poate modifica valorile stocate în cadrul acestora.

Modificarea valorii unui parametru

Pentru a schimba valoarea unui parametru, o funcție trebuie să cunoască adresa de memorie a acelui parametru. Pentru a indica unei funcții adresa unui parametru, programele apelează la operatorul de adresare (&) din C++. Următorul apel de funcție ilustrează modul în care programele folosesc operatorul de adresare pentru a transmite funcției *change_values* adresele variabilelor *big* și *small*.

```
change_values(&big, &small)
```

Transmiterea parametrilor prin adresă

În cadrul definiției funcției, va trebui să informați C++ că programul transmite parametrii către funcție prin adresă. În acest scop veți declara *variabile pointer* prin precedarea fiecărui nume de variabilă cu un asterisc, ca mai jos:

```
void change_values(int *big, int *small)
```

Pointer la tipul int

O variabilă pointer este o variabilă care conține o adresă de memorie. Pentru a modifica apoi valoarea unui parametru în cadrul unei funcții, va trebui să precedați numele parametrului cu un asterisc, așa cum se vede aici:

```
*big = 1001;
*small = 1001;
```

Programul următor, *ChgParam.CPP*, folosește operatorul de adresare (&) pentru a transmite funcției *change_values* adresele parametrilor *big* și *small*. Funcția utilizează, la rândul său, pointeri la locațiile de memorie ale parametrilor. În acest fel, modificările pe care funcția le efectuează asupra parametrilor rămân și după încheierea funcției, așa cum este prezentat în continuare:

```
#include <iostream.h>

void change_values(int *a, int *b)
  {
    *a = 1001;
    *b = 1001;

    cout << "The values within display values are " << *a <<
      " and " << *b << endl;
  }
```

```
void main(void)
  {
    int big = 2002, small = 0;
    cout << "Values before function " << big << " and " <<
      small << endl;
    change_values(&big, &small);
    cout << "Values after function " << big << " and " <<
      small << endl;
  }
```

După compilarea și rularea programului *ChgParam.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> ChgParam <Enter>
Values before function 2002 and 0
The values within display_values are 1001 and 1001
Values after function 1001 and 1001
```

Precum vedeți, valorile pe care funcția *change_values* le atribuie parametrilor se păstrează și după încheierea funcției. Pentru a înțelege de ce se păstrează aceste modificări pe care le efectuează funcția asupra variabilelor, amintiți-vă că funcția are acces la locațiile de memorie ale fiecărei variabile. Atunci când transmiteți parametri către o funcție prin adresă, C++ plasează pe stivă adresa fiecărei variabile, așa cum ilustrează figura 11.2.

Prin utilizarea pointerilor (adresele de memorie) în cadrul funcției, *change_values* poate accesa adresa din memorie a fiecărui parametru, modificând liber valorile.

Modificarea valorilor de parametri în cadrul funcțiilor



Pentru a putea modifica valoarea unui parametru în cadrul unei funcții, respectiva funcție trebuie să cunoască adresa de memorie a acelui parametru. Din acest motiv, programul trebuie să transmită adresa parametrului, folosind în acest scop operatorul de adresare din C++:

```
o_functie(&o_variabila);
```

În cadrul funcției, va trebui să informați C++ că acea funcție va lucra cu o adresă de memorie (un pointer). Pentru aceasta veți preceda numele parametrului din declarație cu un asterisc:

```
void o_functie(int *o_variabila);
```

Pentru a schimba apoi valoarea parametrului în cadrul funcției, va trebui să precedați referințele la numele variabilei pointer cu un asterisc, ca mai jos:

```
*o_variabila = 1001;
cout << *o_variabila;
```

Pentru a preveni erorile, C++ nu va permite unui program să transmită o adresă de variabilă către o funcție care nu așteaptă ca parametru un pointer. Similar, C++ va genera în mod normal un avertisment la compilare atunci când un program încearcă să transmită o valoare către o funcție ce așteaptă ca parametru un pointer.

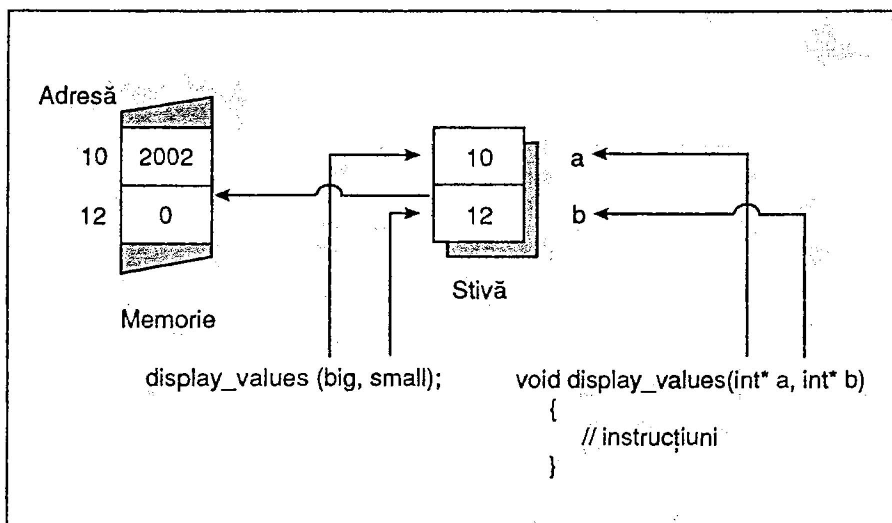


Figura 11.2 *Transmiterea prin adresă a parametrilor unei funcții.*

Un al doilea exemplu

Atunci când programele transmit pointeri la parametri, respectivii parametri pot avea orice tip, precum *int*, *float* sau *char*. Funcția care folosește pointeri va declara variabile de tipul corespunzător, precedând fiecare nume de variabilă cu un asterisc pentru a arăta că respectiva variabilă este un pointer. Programul următor, *SwapVals.CPP*, transmite funcției *swap_values* adresele a doi parametri de tip *float*. Funcția utilizează, la rândul său, pointeri la locațiile de memorie ale fiecărui parametru pentru a inter-schimba valorile parametrilor:

```
#include <iostream.h>

void swap_values(float *a, float *b)
  {
    float temp;

    temp = *a;
    *a = *b;
    *b = temp;
  }

void main(void)
  {
    float big = 10000.0;
    float small = 0.00001;

    swap_values(&big, &small)

    cout << "Big contains " << big << endl;
    cout << "Small contains " << small << endl;
  }
```

După cum puteți vedea, programul transmite parametrii către funcția *swap_values* prin adresă. În cadrul funcției, instrucțiunile utilizează pointeri la adresele de memorie ale fiecărui parametru. Merită să vă opriți mai atent asupra operațiilor efectuate în interiorul funcției *swap_values*. Așa cum se vede, funcția declară *a* și *b* ca pointeri la valori de tip *float*.

```
void swap_values(float *a, float *b)
```

Variabila *temp* este declarată, însă, în funcție ca un simplu *float*, nu ca pointer la *float*:

```
float temp;
```

Să considerăm următoarea instrucțiune:

```
temp = *a;
```

Instrucțiunea determină C++ să atribuie valoarea indicată curent de variabila pointer *a* (care este valoarea *big* – 10000.0) variabilei *temp*. Deoarece *temp* are tipul *float*, atribuirea este corectă. O variabilă pointer este o variabilă care conține o adresă. Instrucțiunea următoare ar declara *temp* ca un pointer la o locație de memorie care conține o valoare de tip *float*:

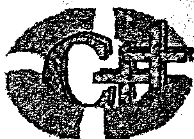
```
float *temp;
```

În acest exemplu, însă, *temp* ar putea reţine doar adresa unei valori în virgulă mobilă, nu şi valoarea propriu-zisă.

Dacă aţi înlătura operatorul de indirectare (*) din faţa variabilei *a* în cadrul operaţiei de atribuire, instrucţiunea ar încerca să atribuie variabilei *temp* valoarea aflată în *a*, care este o adresă. Cum *temp* poate reţine o valoare în virgulă mobilă, nu şi adresa unei valori în virgulă mobilă, în acest caz s-ar produce o eroare.

Dacă nu vă simţiţi prea familiar în ceea ce priveşte pointerii, nu vă faceţi griji, deoarece partea a III-a îi va studia în amănunt. Deocamdată, însă, este suficient să reţineţi că atunci când doriţi ca o funcţie să modifice valoarea unui parametru trebuie să utilizaţi pointeri.

Folosiţi codul în limbaj de asamblare pentru a înţelege modul de funcţionare al compilatorului



Una din cele mai bune căi pentru a înţelege modul în care compilatorul de C++ tratează pointerii este să examinaţi codul generat de compilator în limbaj de asamblare. Cele mai multe compilatoare de C++ dispun de o opţiune în linia de comandă pe care o puteţi folosi la compilarea programului pentru a determina compilatorul să genereze codul în limbaj de asamblare. Prin citirea acestui cod în limbaj de asamblare aţi putea înţelege mai bine modul în care compilatorul utilizează stiva la transmiterea de parametri către funcţii.

Ce trebuie să ştiţi

În lecţia de faţă aţi învăţat să modificaţi valoarea unui parametru în cadrul unei funcţii. Pentru a putea modifica valoarea unui parametru, funcţia trebuie să utilizeze pointeri. La început aţi putea fi uşor intimidat de pointeri. În lecţia 15, „Utilizarea referinţelor C++“, veţi învăţa să folosiţi *referinţe* C++, ceea ce simplifică procesul de modificare a parametrilor în cadrul unei funcţii. Dar pentru că mulţi programatori de C folosesc pointeri pentru modificarea parametrilor, este necesar să cunoaşteţi operaţiile implicate. În lecţia 12, „Utilizarea bibliotecilor de execuţie“, veţi vedea că majoritatea compilatoarelor de C++ oferă biblioteci de funcţii care vă pot economisi un efort de programare considerabil, permiţându-vă să dezvoltaţi rapid programe puternice. Dar înainte de a trece la lecţia 12, asiguraţi-vă că aţi reţinut următoarele aspecte importante:

☑ O funcţie nu poate modifica valoarea unui parametru decât prin intermediul pointerilor sau al referinţelor C++.

☑ Atunci când programul transmite un parametru către o funcție, C++ plasează o copie a valorii parametrului într-o locație de memorie temporară numită stivă. Orice modificări efectuate de funcție asupra parametrului respectiv afectează numai copia valorii aflată pe stivă.

☑ Pentru a modifica valoarea unui parametru, o funcție trebuie să cunoască adresa de memorie a variabilei corespunzătoare.

☑ Programele pot transmite funcțiilor adresele de variabile prin intermediul operatorului de adresare (&) din C++.

☑ Atunci când primește adresa unei variabile, o funcție trebuie să declare variabila parametru ca pointer (prin precedarea numelui de variabilă cu un asterisc).

☑ Atunci când trebuie să utilizeze valoarea referită (indicată) de un pointer, funcția trebuie să preceadă numele variabilei pointer cu un asterisc (*), acesta reprezentând operatorul de indirectare din C++.

Lecția 12

Utilizarea bibliotecilor de execuție

În lecția 10, „O introducere în funcții", ați văzut că puteți să împărțiți programele în fragmente mici și ușor de gestionat care se numesc funcții și care îndeplinesc sarcini bine definite. Unul dintre avantajele utilizării de funcții este faptul că o funcție creată într-un program poate fi adesea folosită și într-un alt program. Așa cum veți afla din lecția de față, majoritatea compilatoarelor de C++ oferă o mulțime cuprinzătoare de funcții ce pot fi utilizate în programele dumneavoastră, mulțime ce constituie *biblioteca de execuție*. Profitând de aceste funcții veți reduce efortul de programare pe care ar trebui altfel să-l depuneți. În schimb, programul va apela pur și simplu o funcție din biblioteca de execuție. În funcție de compilator, biblioteca de execuție poate conține mii de funcții. Această lecție studiază modul de folosire a acestor funcții în cadrul programelor. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- *Biblioteca de execuție* este o mulțime de funcții oferite de compilator care pot fi utilizate lesne în cadrul programelor.
- Pentru utilizarea unei funcții din biblioteca de execuție, este necesară includerea fișierului de antet din biblioteca de execuție care conține prototipul funcției respective.
- Unele compilatoare numesc biblioteca de execuție o *interfață de programare a aplicațiilor* (application programming interface – API).

Majoritatea bibliotecilor de execuție conțin sute de funcții utile care vă pot economisi un timp substanțial la programare, ajutându-vă în dezvoltarea rapidă a unor programe puternice. Așa cum veți vedea, utilizarea funcțiilor din biblioteca de execuție este foarte simplă!

Utilizarea unei funcții din biblioteca de execuție

În lecția 10 ați aflat că, pentru ca un program să poată invoca o funcție, compilatorul de C++ trebuie să fi întâlnit definiția sau prototipul respectivei funcții. Deoarece funcțiile din biblioteca de execuție nu sunt definite în cadrul programului (compilatorul C++ definește funcțiile într-un fișier de bibliotecă), va trebui să specificați prototipul pentru fiecare astfel de funcție pe care intenționați să o folosiți. Pentru a înlesni utilizarea funcțiilor din biblioteca de execuție, compilatorul de C++ pune la dispoziție fișiere de antet care conțin prototipurile de funcții adecvate. În acest fel, este suficient ca programele să includă fișierul de antet corespunzător prin intermediul instrucțiunii *#include* și să apeleze apoi funcția dorită. De exemplu, programul următor, *ShowTime.CPP*, utilizează funcțiile *time* și *ctime* din biblioteca de execuție pentru a afișa data și ora curente. Fișierul de antet *time.h* este cel care conține prototipurile acestor două funcții din biblioteca de execuție, după cum se vede aici:

```
#include <iostream.h>
#include <time.h>       // Pentru functiile din
                        //    biblioteca de executie

void main(void)
  {
    time_t system_time;

    system_time = time(NULL);

    cout << "The current system time is " <<
      ctime(&system_time) << endl;
  }
```

Atunci când compilați și rulați programul *ShowTime.CPP*, pe ecran vor fi afișate data și ora curente, ca mai jos:

```
C:\> ShowTime <Enter>
The current system time is Sat Oct 25 16:13:51 1997
```

După cum vedeți, programul utilizează funcțiile *time* și *ctime*. În cazul funcției *ctime*, programul transmite acesteia adresa variabilei *system_time* prin intermediul operatorului de adresare despre care discutam în lecția 11, „Modificarea valorilor parametrilor". Pentru folosirea acestor funcții din biblioteca de execuție nu a fost necesară decât includerea fișierului de antet *time.h* la începutul fișierului sursă.

Într-un mod similar, programul care urmează, *Sqrt.CPP*, folosește funcția *sqrt* pentru a calcula rădăcina pătrată a mai multor valori diferite. Prototipul pentru funcția *sqrt* se află în fișierul de antet *math.h*, așa cum se poate vedea în continuare:

```
#include <iostream.h>
#include <math.h> // Contine prototipul functiei sqrt

void main(void)
  {
    cout << "The square root of 100.0 is " << sqrt(100.0)
      << endl;
    cout << "The square root of 10.0 is " << sqrt(10.0)
      << endl;
    cout << "The square root of 5. 0 is " << sqrt(5.0)
      << endl;
  }
```

Ca un ultim exemplu, programul *SysCall.CPP* utilizează funcţia *system*, al cărei prototip este conţinut în fişierul de antet *stdlib.h*. Funcţia *system* oferă o cale simplă pentru executarea unei comenzi a sistemului de operare, cum ar fi DIR, sau a unui alt program:

```
#include <stdlib.h>

void main(void)
  {
     system("DIR");
  }
```

În acest exemplu, programul foloseşte funcţia *system* pentru a executa comanda DIR din MS-DOS. Experimentaţi puţin cu acest program şi executaţi şi alte comenzi sau chiar unul dintre programele pe care le-aţi creat mai devreme în această carte.

Despre funcţiile din biblioteca de execuţie

Compilatorul dumneavoastră de C++ dispune în cadrul bibliotecii sale de execuţie de sute de funcţii. Documentaţia care însoţeşte compilatorul ar trebui să conţină o descriere completă a funcţiilor disponibile în biblioteca de execuţie. La parcurgerea acestei documentaţii veţi vedea că funcţiile sunt prezentate, de regulă, prin specificarea prototipului. Spre exemplu, în cazul funcţiei *sqrt* aţi putea întâlni următorul prototip de funcţie:

```
double sqrt(double);
```

În exemplul de mai sus, prototipul funcţiei vă arată că aceasta întoarce o valoare de tip *double* şi aşteaptă un parametru având acelaşi tip *double*. Analog, pentru funcţia *time* aţi putea găsi următorul prototip:

```
time_t time(time_t *);
```

De această dată, prototipul vă informează că funcţia întoarce o valoare de tip *time_t* (pe care îl defineşte fişierul de antet *time.h*). Funcţia aşteaptă ca parametrul primit să fie un pointer la o variabilă de tip *time_t*. Citind informaţii privind funcţiile din biblioteca de execuţie puteţi afla multe lucruri despre funcţii şi despre C++ prin studierea prototipurilor de funcţii.

O altă cale de a afla mai multe despre funcţiile din biblioteca de execuţie a compilatorului este să inspectaţi fişierele de antet care se află în subdirectorul *INCLUDE* din directorul compilatorului. Acordaţi-vă câteva minute pentru a tipări fişierele de antet *math.h*, *ctime.h* şi *stdlib.h* pe care le-au utilizat programele din această lecţie.

Utilizarea funcțiilor API



Pe lângă biblioteca de execuție standard, multe compilatoare pot lucra și cu funcții API (*interfața de programare a aplicațiilor*). Dacă programați, de pildă, sub mediul Windows, aveți la dispoziție funcții API pentru grafică, funcții API pentru telefo- nie (cunoscute ca TAPI), funcții API pentru multimedia și încă multe altele. Înainte de a putea să vă creați propriile funcții, aveți grijă să aflați ce funcții API sunt deja oferite de către compilator.

Ce trebuie să știți

Biblioteca de execuție C++ oferă o mulțime cuprinzătoare de funcții care pot fi utilizate în programe. Încercați să găsiți documentația privitoare la biblioteca de execuție care însoțește compilatorul utilizat. Familiarizați-vă cu funcțiile oferite de această bibliotecă de execuție. Prin utilizarea acestor funcții veți reduce considerabil efortul de programare. În lecția 13, „Variabilele locale și domeniul", veți învăța despre variabile locale și despre domeniu (părțile dintr-un program în care este recunoscut numele unei variabile). Dar înainte de a trece la lecția 13, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Biblioteca de execuție constă într-o mulțime de funcții pe care compilatorul le pune la dispoziția programelor pentru utilizare.

☑ Pentru folosirea unei funcții din biblioteca de execuție trebuie să specificați prototipul funcției respective.

☑ Majoritatea compilatoarelor de C++ dispun de fișiere de antet ce conțin prototipurile corespunzătoare pentru fiecare funcție din biblioteca de execuție.

☑ Pe lângă oferirea unei biblioteci de execuție, multe compilatoare de C++ acceptă și funcții API (interfața de programare a aplicațiilor) care îndeplinesc operații de un anumit gen, cum ar fi programarea pentru grafică sau pentru multimedia.

Lecția 13

Variabilele locale și domeniul

Așa cum ați aflat din lecția 10, „O introducere în funcții", funcțiile vă permit împărțirea programelor în fragmente mici și ușor de gestionat. Toate funcțiile pe care le-ați utilizat până acum au fost destul de simple. Odată ce funcțiile vor efectua activități mai utile, vor exista situații în care acestea vor trebui să utilizeze variabile pentru a-și îndeplini sarcinile. Variabilele declarate în cadrul unei funcții sunt *variabile locale*. Valorile lor și însuși faptul că aceste variabile locale există sunt cunoscute exclusiv de către funcție. Cu alte cuvinte, dacă declarați în funcția *stat_de_plata* o variabilă locală numită *salariu*, nici o altă funcție nu are acces la valoarea *salariu*. De fapt, celelalte funcții nici măcar nu știu că variabila *salariu* există. Lecția de față studiază *domeniul* variabilelor, adică acele părți din program în care o variabilă este recunoscută și poate fi folosită. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Declararea variabilelor locale în cadrul unei funcții se face ca și în cazul programului principal – prin specificarea tipului și numelui variabilei.
- Numele variabilelor folosite într-o funcție trebuie să fie unice numai în cadrul acelei funcții.
- *Domeniul* unei variabile definește părțile din program în care variabila este recunoscută și poate fi accesată.
- Spre deosebire de variabilele locale, *variabilele globale* sunt recunoscute în întregul program și sunt accesibile din orice funcție.
- Operatorul de rezoluție globală (::) din C++ vă permite controlarea domeniului unei variabile.

Declararea variabilelor locale în cadrul unei funcții este foarte simplă. De fapt, acest lucru l-ați făcut deja de fiecare dată când ați declarat variabile în interiorul programului principal.

Declararea variabilelor locale

O *variabilă locală* este o variabilă pe care programul o definește în interiorul unei funcții. Variabila este *locală* acelei funcții deoarece funcția respectivă este singura care știe de existența variabilei și care o poate folosi. Variabilele locale se declară la începutul unei funcții, după acolada care deschide corpul funcției, așa cum se vede aici:

```
void o_functie(void)
  {
    int numar;
    float rezultat;
  }
```

Programul următor, *UseBeeps.CPP*, folosește o funcție *sound_speaker* care generează în difuzorul calculatorului atâtea sunete câte specifică parametrul *beep*. În cadrul funcției *sound_speaker*, variabila locală *counter* urmărește numărul de sunete generate în difuzor de funcție, după cum se vede în continuare:

```
#include <iostream.h>

void sound beeps(int beeps)
  {
    for (int counter = 1; counter <= beeps; counter++)
      cout << '\a';
  }

void main(void)
  {
    sound_beeps(2);
    sound_beeps(3);
  }
```

Precum vedeți, funcția *sound_beep* declară variabila *counter* imediat după acolada ce deschide corpul funcției. Deoarece *counter* este definită în funcția *sound_beeps*, această variabilă este locală funcției *sound_beeps*, ceea ce înseamnă că numai *sound_beeps* recunoaște și poate accesa variabila în cauză.

Despre conflictele de nume

Atunci când declarați variabile locale în cadrul funcțiilor, este foarte posibil ca numele uneia dintre variabilele locale declarate într-o funcție să fie identic cu un nume de variabilă utilizat într-o altă funcție. Așa cum am menționat, o variabilă locală este cunoscută exclusiv de funcția în care sunt specificate tipul și numele acelei variabile. Din această cauză, dacă două funcții utilizează un același nume de variabilă locală nu apare nici un conflict. C++ tratează fiecare nume de variabilă ca fiind local funcției corespunzătoare.

Programul următor, *LclName.CPP*, utilizează funcția *add_values* pentru a aduna două numere întregi. Funcția atribuie rezultatul variabilei locale *value*. În programul principal, însă, unul dintre parametrii transmiși funcției este de asemenea numit *value*. Dar cum C++

tratează cele două variabile ca locale funcțiilor corespunzătoare, numele nu intră în conflict, așa cum se vede în continuare:

```
#include <iostream.h>

int add_values(int a, int b)
  {
    int value;

    value = a + b;

    return(value);
  }

void main(void)
  {
    int value = 1001;
    int other_value = 2002;

    cout << value << " + " << other_value << " = " <<
      add_values(value, other_value) << endl;
  }
```

Despre variabilele locale



Variabilele locale sunt variabile declarate în cadrul unei funcții. Numele și valorile variabilelor locale sunt cunoscute exclusiv funcției care le declară. Variabilele locale ar trebui declarate la începutul funcțiilor, imediat după acolada ce deschide corpul funcției. Numele alese pentru variabilele locale trebuie să fie unice în interiorul funcțiilor corespunzătoare. La declararea unei variabile locale într-o funcție, variabila poate fi inițializată prin intermediul operatorului de atribuire.

Despre variabilele globale

După cum ați văzut, o variabilă locală este declarată în interiorul unei anumite funcții și este cunoscută exclusiv de către aceasta. Pe lângă variabilele locale, C++ permite programelor să declare *variabile globale*, care sunt recunoscute oriunde în program (globale tuturor funcțiilor). Pentru a declara o variabilă globală nu trebuie decât să plasați declarația variabilei respective la începutul programului, în afara oricărei funcții, ca mai jos:

```
int o_variabila_globala;  ├────────── Declarația unei variabile globale

void main(void)
  {
    // Aici se afla instructiunile programului
  }
```

Programul următor, *Global.CPP*, utilizează o variabilă globală numită *number*. Oricare dintre funcțiile programului pot folosi (sau modifica) valoarea acestei variabile globale. În exemplul nostru, fiecare funcție afișează valoarea curentă a variabilei și apoi o incrementează cu 1:

```
#include <iostream.h>

int number = 1001;

void first_change(void)
  {
    cout << "number's value in first_change " << number
      << endl;
    number++;
  }

void second_change(void)
  {
    cout << "number's value in second_change " << number
      << endl;
    number++;
  }

void main(void)
  {
    cout << "number's value in main " << number << endl;
    number++;
    first_change();
    second_change();
  }
```

După compilarea și rularea programului *Global.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> Global <Enter>
number's value in main 1001
number's value in first_change 1002
number's value in second_change 1003
```

Ca o regulă generală, este bine să evitați utilizarea în programe a variabilelor globale. Deoarece valoarea unei variabile globale poate fi modificată de orice funcție, este dificil de urmărit toate funcțiile în care este posibil ca valoarea respectivă să fie alterată. În schimb, programele ar trebui să declare variabilele în funcția *main* și apoi să le transmită (ca parametri) acelor funcții care le necesită. (Nu uitați că C++ plasează pe stivă o copie temporară a valorii unei variabile, lăsând originalul neschimbat.)

Când apar conflicte între numele variabilelor globale și cele ale variabilelor locale

Este bine să evitați utilizarea variabilelor globale oricând este posibil. Dacă un program trebuie, însă, să utilizeze o variabilă globală, ar putea fi cazuri în care numele acelei variabile intră în conflict cu numele unei variabile locale. Atunci când apare un astfel de conflict, C++ acordă prioritate variabilei locale. Cu alte cuvinte, programul presupune că fiecare referire la numele aflat în conflict corespunde variabilei locale.

Ar putea fi, totuși, situații în care devine necesară accesarea unei variabile globale al cărei nume intră în conflict cu o variabilă locală. În astfel de situații, utilizarea variabilei globale se poate face prin intermediul *operatorului de rezoluție globală* (::) din C++. Spre exemplu, să presupunem că avem o variabilă locală și una globală care sunt numite ambele *number*. Atunci când doriți utilizarea variabilei locale *number* într-o funcție, este suficient să specificați numele variabilei, ca aici:

```
number = 1001;      // Referinta la variabila locala
```

Atunci când vreți, însă, ca funcția să folosească variabila globală, apelați la operatorul de rezoluție globală, așa cum se vede aici:

```
::number = 2002;      // Referinta la variabila globala
```

Programul următor, *GlobLoca.CPP*, utilizează variabila globală *number*. În plus, funcția *show_numbers* folosește o variabilă locală numită *number*. Pentru accesarea variabilei globale, funcția apelează la operatorul de rezoluție globală, ca mai jos:

```
#include <iostream.h>

int number = 1001; // Variabila globala

void show_numbers(int number)
  {
    cout << "Local variable number contains " << number
      << endl;
    cout << "Global variable number contains "
      << ::number << endl;
  }

void main(void)
  {
    int some_value = 2002;

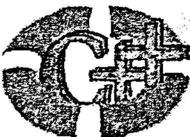
    show_numbers(some_value);
  }
```

La compilarea și rularea programului *GlobLoca.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> GlobLoca <Enter>
Local variable number contains 2002
Global variable number contains 1001
```

După cum puteți vedea, programele pot selecta fie variabila globală, fie pe cea locală, folosind operatorul de rezoluție globală. Și totuși, așa cum ați remarcat probabil, utilizarea variabilelor globale alături de cele locale poate genera confuzii, ceea ce poate duce apoi la erori. Prin urmare, este bine să evitați pe cât posibil utilizarea variabilelor globale.

Despre variabilele globale



O variabilă globală este o variabilă ale cărei nume și valoare sunt cunoscute oriunde în program. Pentru a crea o variabilă globală, aceasta trebuie declarată înspre începutul fișierului sursă și în exteriorul oricărei funcții. Variabila globală poate fi utilizată de toate funcțiile care urmează declarația respectivei variabile. Deoarece, însă, utilizarea neglijentă a variabilelor globale poate duce la apariția de erori, este bine să evitați folosirea acestora ori de câte ori este posibil.

Despre domeniul unei variabile

Dacă citiți cărți și articole din reviste referitoare la C++ este posibil să întâlniți termenul *domeniu*, care definește acele părți din program în care numele unei variabile are sens (și poate fi, în consecință, utilizat). În cazul unei variabile locale, C++ restrânge domeniul variabilei la funcția în care aceasta este declarată. Variabilele globale, pe de altă parte, sunt recunoscute oriunde în program. Așadar, variabilele globale au un domeniu mai întins.

Ce trebuie să știți

În lecția de față ați învățat să declarați variabile locale în cadrul funcțiilor. Pe măsură ce funcțiile îndeplinesc operații tot mai utile, ele vor necesita prezența variabilelor locale. Această lecție v-a prezentat, de asemenea, variabilele globale, ale căror nume și valori sunt cunoscute pe toată întinderea programului. Deoarece variabilele globale pot provoca erori care sunt dificil de identificat, este bine să evitați utilizarea acestor variabile ori de câte ori este posibil. În lecția 14, „Supradefinirea funcțiilor", veți vedea că C++ permite programelor să declare două sau mai multe funcții având un același nume, dar primind parametri diferiți sau întorcând valori de tipuri diferite. Printr-o astfel de „supradefinire" a unui nume de funcție puteți simplifica utilizarea funcțiilor. Dar înainte de a trece la lecția 14, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Variabilele locale sunt variabilele pe care programul le declară în interiorul funcțiilor.
- ☑ Variabilele locale sunt recunoscute exclusiv de funcțiile care le declară.
- ☑ Două sau mai multe funcții pot folosi același nume de variabilă locală fără a apărea nici un conflict.
- ☑ O variabilă globală este o variabilă ale cărei nume și valoare sunt cunoscute în tot programul.
- ☑ Pentru a crea o variabilă globală, declarați variabila spre începutul fișierului sursă, în exteriorul oricărei funcții.
- ☑ Deoarece valoarea unei variabile globale poate fi ușor modificată de orice funcție, variabilele globale fac posibilă apariția în programe a unor erori greu de detectat. Din acest motiv este bine să evitați utilizarea variabilelor globale.

Lecția 14

Supradefinirea funcțiilor

La definirea unei funcții într-un program trebuie să specificați tipul întors de funcție și numărul și tipurile parametrilor. Dacă ați fi programat înainte în limbajul C și ați fi avut o funcție numită *aduna_numere* care lucra cu două numere întregi, în cazul în care intenționați să utilizați o funcție similară pentru a aduna trei numere întregi ar fi fost necesar să creați o funcție cu un alt nume. Atunci ați fi putut utiliza funcțiile *aduna_doua_valori*, *aduna_trei_valori* și tot așa. Similar, dacă vroiați să folosiți o funcție care să adune două valori de tip *float*, aveați nevoie de o altă funcție cu numele său propriu.

Pentru a elimina această duplicare a funcțiilor, C++ vă permite să definiți mai multe funcții cu același nume. La compilarea programului, compilatorul de C++ examinează numărul de parametri pe care îi primește fiecare funcție și apelează apoi funcția corespunzătoare. Procesul de oferire a mai multor funcții în vederea unei selecții a compilatorului se numește *supradefinire*. Lecția de față se oprește asupra modului de supradefinire a funcțiilor în cadrul programelor. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Supradefinirea funcțiilor vă permite folosirea aceluiași nume de funcție cu tipuri diferite ale parametrilor sau cu numere diferite de parametri.
- Supradefinirea funcțiilor în cadrul unui program se face prin simpla definire a două funcții cu același nume și același tip al valorii întoarse, dar care diferă prin numărul sau tipurile parametrilor.

Supradefinirea funcțiilor este o facilitate C++ pe care limbajul de programare C nu o oferea. Așa cum veți vedea, supradefinirea funcțiilor este convenabilă și poate spori lizibilitatea programelor.

O introducere în supradefinirea funcțiilor

Supradefinirea funcțiilor permite programelor definirea mai multor funcții care au același nume și același tip al valorii întoarse. De exemplu, programul următor, *Overload.CPP*, supradefinește funcția *add_values*. Prima funcție definită adună două valori de tip *int*, în timp ce a doua funcție definită adună trei valori. În timpul compilării, compilatorul de C++ determină funcția care trebuie utilizată:

```
#include <iostream.h>

int add_values(int a, int b)
  {
    return(a + b);
  }

int add_values(int a, int b, int c)
  {
    return(a + b + c);
  }

void main(void)
  {
    cout << "200 + 801 = " << add_values(200, 801) << endl;
    cout << "100 + 201 + 700 = " << add_values(100, 201, 700)
      << endl;
  }
```

După cum puteţi vedea, programul defineşte două funcţii numite *add_values*. Prima funcţie adună două valori de tip *int*, în timp ce a doua funcţie adună trei valori. Nu trebuie să faceţi nimic deosebit pentru a avertiza compilatorul despre supradefinire. Este suficient să definiţi şi să utilizaţi funcţiile. Compilatorul, în schimb, va realiza ce definiţie de funcţie trebuie utilizată, în funcţie de parametrii transmişi de program. Într-o manieră similară, programul următor, *Msg_Ovr.CPP*, supradefineşte funcţia *show_message*. Prima funcţie afişează un mesaj implicit atunci când programul nu precizează nici un parametru. Cea de a doua funcţie afişează mesajul pe care programul îl transmite funcţiei, iar a cea de a treia funcţie afişează două mesaje:

```
#include <iostream.h>

void show_message(void)
  {
    cout << "Default message: Rescued by C++" << endl;
  }

void show_message(char *message)
  {
    cout << message << endl;
  }
```

```
void show_message(char *first, char *second)
  {
    cout << first << endl;
    cout << second << endl;
  }

void main(void)
  {
    show_message();
    show_message("I've Been Rescued!");
    show_message("C++ is not so hard!", " Overloading is
    cool!");
  }
```

Când trebuie folosită supradefinirea

Una din utilizările cele mai frecvente ale supradefinirii este în cazul folosirii unei funcții în vederea obținerii unui rezultat, chiar dacă tipurile parametrilor diferă. De exemplu, să presupunem că un program conține o funcție numită *ziua_saptamanii* care întoarce ziua curentă din săptămână (0 pentru duminică, 1 pentru luni și așa mai departe, până la 6 pentru sâmbătă). Programul ar putea supradefini această funcție astfel încât să întoarcă în mod corect ziua din săptămână atât atunci când programul îi transmite ca parametru o zi din calendarul Iulian, cât și atunci când programul transmite funcției ziua, luna și anul curente, ca în continuare:

```
int ziua_saptamanii(int zi_Iulian)

  {
    // Instructiuni
  }

int ziua_saptamanii(int zi, int luna, int an)
  {
    // Instructiuni
  }
```

Când veți studia în cadrul lecțiilor următoare facilitățile de programare orientată spre obiect din C++, veți folosi supradefinirea funcțiilor pentru a spori capacitățile programelor.

Supradefinirea funcțiilor crește lizibilitatea programelor



Supradefinirea funcțiilor din C++ permite programelor să definească mai multe funcții cu același nume. Funcțiile supradefinite trebuie să difere prin numărul și tipurile parametrilor. Înainte de apariția în C++ a supradefinirii funcțiilor, programele C erau nevoite să includă mai multe funcții având nume similare. Din păcate, programatorii care foloseau acele funcții trebuiau să-și amintească ce funcție corespundea fiecărei combinații de parametri. Supradefinirea funcțiilor, pe de altă parte, simplifică sarcina programatorilor, necesitând reținerea unui singur nume de funcție.

Ce trebuie să știți

Supradefinirea funcțiilor permite programelor să specifice mai multe definiții pentru o singură funcție. La compilarea unui program, compilatorul de C++ va determina funcția care trebuie folosită pe baza numărului și tipurilor parametrilor pe care programul îi transmite funcției. În lecția de față ați văzut cât de ușor pot fi supradefinite funcțiile din cadrul programelor. În lecția 15, „Utilizarea referințelor C++", veți vedea cum pot referințele C++ să simplifice procesul de modificare a parametrilor în cadrul unei funcții. Dar înainte de a trece la lecția 15, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Supradefinirea funcțiilor permite programelor să ofere mai multe „perspective" asupra unei aceleiași funcții din program.

☑ Pentru supradefinirea unei funcții în cadrul unui program nu trebuie decât să definiți două sau mai multe funcții având același nume, dar care diferă prin numărul sau tipurile parametrilor acceptați.

☑ În timpul compilării, compilatorul de C++ determină ce funcție supradefinită trebuie apelată, în funcție de numărul și tipurile parametrilor pe care programul îi transmite funcției.

☑ Supradefinirea funcțiilor simplifică procesul de programare prin aceea că permite programatorilor să lucreze cu un singur nume de funcție atunci când programele trebuie să îndeplinească o anumită sarcină.

Lecția 15

Utilizarea referințelor C++

În lecția 11, „Modificarea valorilor parametrilor", ați învățat să modificați în cadrul funcțiilor valorile de parametri prin intermediul pointerilor. Așa cum ați văzut, pentru utilizarea pointerilor trebuie să precedați numele variabilelor pointer cu un asterisc. Utilizarea pointerilor în cadrul programelor C++ este o moștenire ce provine din limbajul C. Pentru a simplifica procesul de modificare a parametrilor în cadrul unei funcții, C++ pune la dispoziție *referințele*. După cum veți învăța în lecția de față, o referință reprezintă un alias (adică un nume alternativ) pe care programele îl pot utiliza pentru a referi o variabilă. Prin folosirea într-o funcție a unei referințe la o variabilă puteți să modificați valoarea unui parametru fără a mai recurge la pointeri și la notațiile corespunzătoare. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru a declara și inițializa o referință în cadrul unui program, declarați o variabilă, plasați un ampersand (&) imediat după tipul variabilei și apoi folosiți operatorul de atribuire pentru a stabili un alias, ca de pildă *int& nume_alias = variabilă;*.
- Programele pot transmite funcțiilor o referință pe post de parametru, iar funcțiile pot modifica astfel valoarea parametrului corespunzător fără a utiliza pointeri.
- În cadrul unei funcții, declararea unui parametru ca referință se face prin plasarea unui ampersand (&) după tipul parametrului – după care veți putea modifica valoarea parametrului în interiorul funcției fără a mai utiliza pointeri.

În cazul funcțiilor care folosesc pointeri, programatorii de C++ începători sunt adesea derutați despre când trebuie folosit asteriscul (*) pentru a „indirecta" un pointer și când trebuie precedat numele unei variabile de operatorul de adresare (&). Așa cum veți vedea, utilizarea referințelor face ca modificarea valorilor de parametri în cadrul funcțiilor să devină foarte simplă.

O referință este un alias

O *referință* C++ permite programelor să creeze un alias (adică un nume alternativ) pentru o variabilă a programului. Declararea unei referințe în program se face prin specificarea unui tip, urmat imediat de caracterul ampersand (&). La declararea unei referințe trebuie să indicați imediat variabila pentru care referința va constitui un alias, așa cum se vede aici:

```
int& nume_alias = variabila;
```

Declarația unei referințe

După declararea unei referințe, programul poate folosi la fel de bine variabila sau referința, după cum se arată mai jos:

```
nume_alias = 1001;

variabila = 1001;
```

Programul următor, *Show_Ref.CPP*, creează o referință numită *alias_name* și asociază acestui alias variabila *number*. Programul folosește apoi atât referința, cât și variabila, ca în continuare:

```
#include <iostream.h>

void main(void)
  {
    int number = 501;
    int& alias_name = number; // Creeaza o referinta

    cout << "The variable number contains " << number
      << endl;
    cout << "The alias to number contains " << alias_name
      << endl;

    alias_name = alias_name + 500;

    cout << "The variable number contains " << number
      << endl;
    cout << "The alias to number contains " << alias_name
      << endl;
  }
```

După cum puteți vedea, programul adună la referința *alias_name* valoarea 500. În consecință, programul adună de fapt 500 la variabila *number* corespunzătoare pentru care referința reprezintă un alias, sau un nume alternativ. La compilarea și rularea programului *Show_Ref.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> Show_Ref <Enter>
The variable number contains 501
The alias to number contains 501
The variable number contains 1001
The alias to number contains 1001
```

Ca o regulă generală, utilizarea unei referințe, așa cum tocmai am ilustrat, face ca programele să fie dificil de înțeles. Veți vedea, însă, că utilizarea unei referințe poate în schimb, să înlesnească substanțial operația de modificare într-o funcție a valorii unui parametru.

Declararea unei referințe în cadrul programului



O referință C++ este un alias (adică un nume alternativ) pe care programele îl pot utiliza pentru a referi o variabilă. Pentru declararea unei referințe, plasați caracterul ampersand (&) imediat după tipul variabilei și apoi specificați numele referinței, urmat de semnul egal și de numele variabilei pentru care referința constituie un alias, ca mai jos:

```
float& alias_salariu = salariu;
```

Utilizarea referințelor ca parametri

Scopul de bază al unei referințe este să simplifice procesul de modificare în cadrul unei funcții a valorilor de parametri. Programul care urmează, *Referenc.CPP*, creează pentru variabila *number* o referință numită *number_alias*. Programul transmite variabila referință către funcția *change_value*, iar aceasta atribuie variabilei valoarea 1001, așa cum se vede în continuare:

```
#include <iostream.h>

void change_value(int &alias)
  {
    alias = 1001;
  }

void main(void)
  {
    int number;
    int& number_alias = number;

    change_value(number_alias);

    cout << "The variable number contains " << number
      << endl;
  }
```

După cum puteți observa, programul transmite referința funcției *change_value*. Dacă priviți declarația funcției, veți vedea că *change_value* declară parametrul *alias* ca o referință la o valoare de tip *int*:

```
void change_value(int& alias)
```

Codul funcției *change_value* poate modifica valoarea parametrului fără a mai trebui să utilizeze un pointer. Prin urmare, codul funcției nu mai folosește asteriscul (*), iar operațiile efectuate de către funcție devin mai ușor de înțeles.

Folosiți comentarii pentru a explica referințele create în programe



Majoritatea programatorilor de C++ sunt obișnuiți cu limbajul C și cu utilizarea pointerilor în cadrul funcțiilor care trebuie să modifice valoarea unui parametru. Ca o consecință, atunci când acești programatori nu văd nici un pointer în funcțiile care folosesc referințe, ei ar putea presupune că valorile parametrilor nu se modifică. Pentru a evita asemenea confuzii, aveți grijă să plasați mai multe comentarii înainte și după funcțiile care modifică parametrii prin intermediul referințelor. În acest fel, programatorii de C vor înțelege corespunzător modul de operare al acelor funcții.

Studiul unui al doilea exemplu

În lecția 11 ați folosit pentru inter-schimbarea a două valori în virgulă mobilă următoarea funcție:

```
void swap_values(float *a, float *b)
  {
    float temp;
    temp = *a;
    *a = *b;
    *b = temp;
  }
```

După cum puteți vedea, funcția combină variabilele pointer cu cele normale. Programul următor, *Swap_Ref.CPP*, simplifică funcția prin folosirea referințelor la valori de tip virgulă mobilă:

```
#include <iostream.h>

void swap values(float& a, float& b)
  {
    float temp;
```

```
        temp = a;
        a = b;
        b = temp;
    }

void main(void)
    {
        float big = 10000.0;
        float small = 0.00001;
        float& big_alias = big;
        float& small_alias = small;

        swap values(big_alias, small_alias);
        cout << "Big contains " << big << endl;
        cout << "Small contains " << small << endl;
    }
```

Precum vedeți, atunci când programul utilizează referințe la valori de tip virgulă mobilă, funcția *swap_values* devine mai ușor de înțeles. În schimb, programul conține acum două noi nume (referințele *big_alias* și *small_alias*) pe care trebuie să le gestionați.

Reguli pentru utilizarea referințelor

O referință nu este o variabilă. După ce asociați o variabilă unei referințe, acea referință nu mai poate fi modificată. În plus, spre deosebire de pointeri, asupra referințelor nu puteți efectua următoarele operații:

- Nu puteți obține adresa unei referințe prin intermediul operatorului de adresare din C++.
- Nu puteți atribui un pointer unei referințe.
- Nu puteți compara valorile referințelor prin intermediul operatorilor relaționali din C++.
- Nu puteți efectua asupra unei referințe operații aritmetice, precum adunarea unui deplasament.
- Nu puteți modifica o referință.

Vă veți întâlni din nou cu referințele atunci când programele dumneavoastră vor utiliza facilitățile de programare orientată spre obiect din C++.

Utilizarea referinţelor pentru modificarea parametrilor unei funcţii



În lecţia 11 aţi văzut că programele pot modifica valorile de parametri în cadrul funcţiilor prin folosirea pointerilor. Pentru ca o funcţie să poată modifica un parametru, respectivei funcţii îi trebuie transmisă adresa acelui parametru. Adresa unui parametru se obţine cu ajutorul operatorului de adresare (&) din C++. Funcţia, la rândul său, foloseşte variabile pointer (care conţin adrese de memorie). Declararea unei variabile pointer în cadrul unei funcţii se face prin precedarea numelui de parametru cu un asterisc (*). Pentru a modifica sau utiliza valoarea parametrului în cadrul funcţiei, trebuie să precedaţi fiecare referire la numele parametrului cu operatorul de indirectare (*) din C++. Din păcate, multe din operaţiile funcţiilor combină variabile pointer şi variabile normale.

Referinţele C++ simplifică procesul de modificare a parametrilor prin eliminarea acelor instrucţiuni care combină variabilele pointer şi cele normale. Preţul care trebuie plătit la utilizarea referinţelor îl reprezintă variabilele suplimentare pe care un programator care parcurge codul va trebui să le recunoască şi să le înţeleagă.

Ce trebuie să ştiţi

În capitolul de faţă aţi aflat cum puteţi folosi referinţele C++ pentru a crea un alias (adică un nume alternativ) pentru o variabilă. Utilizarea referinţelor poate simplifica funcţiile care modifică valorile parametrilor. În lecţia 16, „Precizarea valorilor implicite pentru parametri", veţi vedea că C++ vă permite specificarea de valori implicite pentru parametrii funcţiilor. Dacă un program omite una sau mai multe valori de parametri la apelul unei funcţii, acea funcţie va folosi valorile implicite. Dar înainte de a trece la lecţia 16, asiguraţi-vă că aţi reţinut următoarele aspecte importante:

- ☑ O referinţă C++ este un alias (adică un nume alternativ) pentru o variabilă.
- ☑ Pentru a declara o referinţă, plasaţi caracterul ampersand (&) imediat după tipul variabilei şi apoi specificaţi numele referinţei, urmat de semnul egal şi de numele variabilei pentru care referinţa constituie un alias.
- ☑ După asocierea unei referinţe cu o variabilă, valoarea respectivei referinţe nu mai poate fi modificată.
- ☑ Este bine să inseraţi mai multe comentarii înaintea şi în interiorul funcţiilor care utilizează referinţe la modificarea valorilor de parametri pentru a vă asigura că un alt programator care citeşte codul va sesiza respectivele modificări.
- ☑ Utilizarea excesivă a referinţelor poate duce la un cod de program dificil de înţeles, atât pentru dumneavoastră, cât şi pentru alţi programatori.

Lecția 16

Precizarea valorilor implicite pentru parametri

Așa cum ați învățat, C++ permite programelor să transmită informații funcțiilor sub formă de parametri. În lecția 14, „Supradefinirea funcțiilor", ați văzut că C++ vă permite supradefinirea funcțiilor prin specificarea de definiții care acceptă numere diferite de parametri sau chiar tipuri diferite ale parametrilor. În plus, C++ permite programelor să omită parametri la apelul unei funcții. În asemenea cazuri, C++ utilizează pentru parametrii absenți anumite valori implicite. Lecția de față se oprește asupra modului de definire a valorilor implicite pentru parametri în cadrul funcțiilor. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- C++ permite programelor să precizeze valori implicite ale parametrilor.
- Valorile implicite ale parametrilor se specifică în antetul funcției, la definirea acesteia.
- Dacă un apel de funcție omite unul sau mai mulți parametri, C++ va utiliza pentru aceștia valorile implicite.
- Atunci când un apel de funcție omite un parametru, toți parametrii care urmează trebuie de asemenea omiși.

Precizarea de valori implicite pentru parametri simplifică utilizarea în program a funcțiilor. În consecință, o funcție va putea fi folosită de mai multe programe – ceea ce sporește *reutilizabilitatea* funcției.

Specificarea valorilor implicite

Specificarea de valori implicite pentru parametrii unei funcții se face foarte ușor. Pe scurt, veți folosi operatorul de atribuire din C++ pentru a atribui o valoare unui parametru care apare în definiția funcției, așa cum se vede în continuare:

```
void o_functie(int dimensiune=12, float pret=19.95)
  {
    // Instructiunile functiei
  }
```

Programul următor, *Defaults.CPP*, atribuie valori implicite parametrilor *a*, *b* și *c* din funcția *show_parameters*. Programul apelează apoi funcția, mai întâi fără nici un parametru, specificând după aceea o valoare pentru *a*, apoi valori pentru *a* și *b* și, în fine, valori pentru toți cei trei parametri:

```
#include <iostream.h>

void show_parameters(int a=1, int b=2, int c=3)
  {
    cout << "a " << a << " b " << b << " c " << c << endl;
  }

void main(void)
  {
    show_parameters();
    show_parameters(1001);
    show_parameters(1001, 2002);
    show_parameters(1001, 2002, 3003)
  }
```

După compilarea și rularea programului *Defaults.CPP*, pe ecran vor fi afișate următoarele:

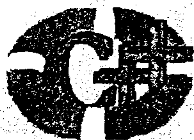
```
C:\> Defaults <Enter>
a 1 b 2 c 3
a 1001 b 2 c 3
a 1001 b 2002 c 3
a 1001 b 2002 c 3003
```

După cum vedeți, funcția folosește corespunzător valorile implicite ale parametrilor.

Reguli privind omiterea valorilor de parametri

Atunci când un program omite parametri ai unei funcții care oferă valori implicite, respectivul program trebuie să omită toți parametrii care urmează. Cu alte cuvinte, nu puteți omite un parametru din mijloc. În cazul programului anterior, dacă doriți omiterea valorii pentru parametrul *b* în apelul *show_parameters*, atunci trebuie să omiteți și valoarea pentru *c*. Nu puteți specifica valori pentru *a* și *c*, omițând valoarea lui *b*.

Specificarea valorilor implicite pentru parametri



La definirea unei funcții, C++ vă permite precizarea de valori implicite pentru unul sau mai mulți parametri. Dacă programul va omite mai apoi unul sau mai mulți parametri la apelul funcției, aceasta va utiliza valorile implicite. Pentru a atribui o valoare implicită unui parametru este suficient să folosiți operatorul de atribuire în cadrul definiției funcției. Spre exemplu, funcția *salariu* de mai jos specifică valori implicite pentru parametrii *ore* și *pe_ora*:

```
float salariu(int id_angajat, float ore = 40, float
  pe_ora = 5.50)
  {
    // instructiuni
  }
```

Atunci când omite un anumit parametru din apelul unei funcții, un program trebuie să omită valorile tuturor parametrilor care urmează.

Ce trebuie să știți

În cuprinsul lecției de față ați aflat că C++ vă permite să precizați valori implicite pentru parametrii funcțiilor. Dacă un program omite unul sau mai mulți parametri din apelul funcției, aceasta va utiliza valorile implicite corespunzătoare. În lecțiile următoare, atunci când veți începe să folosiți în programe facilitățile de programare orientată spre obiect din C++, veți folosi parametrii impliciți pentru a inițializa diferite variabile ale unei clase. După cum ați învățat, o variabilă vă permite păstrarea unei valori de un anumit tip (*int*, *float* și așa mai departe). În lecția 17, „Utilizarea constantelor și a macrodefinițiilor", veți vedea cum utilizarea constantelor și a macrodefinițiilor poate să înlesnească efortul de programare și să ducă la un cod mai lizibil. Dar înainte de a trece la lecția 17, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Pentru a defini valori implicite ale parametrilor de funcții, folosiți operatorul de atribuire din C++ pentru a atribui o valoare unui astfel de parametru în definiția funcției.
- ☑ Atunci când un program omite valori de parametri la apelul unei funcții, acea funcție va utiliza valorile implicite corespunzătoare.
- ☑ Atunci când omite valoarea unui parametru, un program trebuie să omită valorile tuturor parametrilor care urmează – programele nu pot omite valoarea unui parametru din mijloc.
- ☑ Prin precizarea valorilor implicite pentru parametri, puteți să faceți funcțiile mai ușor de folosit și, eventual, mai potrivite pentru utilizarea de către alți programatori.

Lecția 17

Utilizarea constantelor și a macrodefinițiilor

Pe măsură ce programele dumneavoastră vor crește în complexitate, ele ar putea deveni totodată din ce în ce mai dificile de înțeles pentru alți programatori. Pentru a spori lizibilitatea programelor, C++ permite utilizarea constantelor denumite și a macrodefinițiilor. Prin utilizarea constantelor denumite puteți înlocui în codul sursă o valoare numerică, așa cum este 50, cu o constantă sugestivă, precum *DIMENSIUNE_CLASA*. Atunci când un alt programator va citi codul, acesta nu va fi nevoit să ghicească ce reprezintă valoarea numerică 50. În schimb, de fiecare dată când va întâlni *DIMENSIUNE_CLASA*, acel programator va ști că este vorba de o valoare care corespunde numărului de elevi dintr-o clasă. Similar, prin intermediul *macrodefinițiilor*, programele pot înlocui formule complexe ca cea de mai jos cu un nume sugestiv de macrodefiniție:

```
rezultat = (x*y-3) * (x*y-3) * (x*y-3);
```

Prin utilizarea unei macrodefiniții, programele pot înlocui această formulă complexă cu o macrodefiniție numită *CUB* asemănătoare unei funcții, așa cum se vede aici:

```
rezultat = CUB(x*y-3);
```

În acest exemplu, macrodefiniția nu numai că sporește lizibilitatea codului, ci simplifică totodată instrucțiunea, reducând șansele de apariție a erorilor. Lecția de față studiază în detaliu constantele denumite și macrodefinițiile. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru a vă face programele mai ușor de citit, înlocuiți valorile numerice cu constante denumite sugestive.
- Utilizarea în programe a constantelor denumite în locul valorilor numerice poate înlesni modificarea ulterioară a programelor.
- C++ permite programelor să înlocuiască formulele cu nume sugestive de macrodefiniții.
- Înainte de a compila un program, compilatorul de C++ apelează la un program special, numit *preprocesor*, pentru a înlocui fiecare constantă denumită și fiecare macrodefiniție cu valoarea corespunzătoare.
- Macrodefinițiile se execută mai repede decât funcțiile, dar, de asemenea, sporesc dimensiunea programului executabil.
- Majoritatea compilatoarelor de C++ definesc propriile constante și macrodefiniții pe care le puteți utiliza în programe.

Utilizarea constantelor denumite

O *constantă denumită* nu este altceva decât un nume căruia îi asociați o valoare constantă, valoare care, spre deosebire de cea a unei variabile, nu se poate modifica pe durata rulării programului. Constantele denumite se creează prin folosirea directivei preprocesor *#define* (o directivă este o instrucțiune specială pentru preprocesorul compilatorului). Spre exemplu, instrucțiunea următoare definește constanta denumită *DIMENSIUNE_CLASA* ca având valoarea 50:

```
#define DIMENSIUNE_CLASA 50
```

Pentru a deosebi constantele denumite de variabile, majoritatea programatorilor folosesc pentru constantele denumite litere mari. De exemplu, programul următor, *Constant.CPP*, definește și afișează constanta denumită *CLASS_SIZE*:

```
#include <iostream.h>

#define CLASS_SIZE 50 // Numarul de elevi dintr-o clasa

void main(void)
  {
    cout << "CLASS SIZE constant is " << CLASS_SIZE << endl;
  }
```

După cum puteți vedea, programul definește constanta prin plasarea directivei *#define* în partea de început a codului sursă. După definirea unei constante, valoarea acesteia poate fi folosită oriunde în program prin simpla specificare a numelui respectivei constante.

Notă: *Definițiile de constante denumite nu se încheie cu punct și virgulă. În cazul în care plasați punct și virgulă la sfârșitul unei definiții, preprocesorul va include acest semn în cadrul valorii. De exemplu, dacă ați plasa punct și virgulă după valoarea 50 din directiva* ***#define*** *a programului de mai sus, preprocesorul ar înlocui ulterior fiecare apariție a constantei* ***CLASS_SIZE*** *cu valoarea 50 urmată de punct și virgulă (50;), ceea ce foarte probabil ar genera o eroare de sintaxă.*

Despre directivele preprocesor



Înainte de a efectua compilarea unui program, compilatorul de C++ rulează un program special numit preprocesor. Preprocesorul caută în program liniile care încep cu un #, așa cum sunt *#include* sau *#define*. Dacă întâlnește, de pildă, o directivă *#include*, preprocesorul va include fișierul specificat în fișierul sursă, ca și cum conținutul acelui fișier ar fi fost tastat odată cu crearea codului sursă. Toate programele pe care le-ați creat în această carte au utilizat o directivă *#include* pentru a determina preprocesorul să includă conținutul fișierului de antet *iostream.h* în fișierul sursă. Atunci când întâlnește o directivă *#define*, preprocesorul creează o constantă denumită sau o macrodefiniție. Ulterior, la întâlnirea numelui unei constante sau al unei macrodefiniții, preprocesorul va înlocui respectivul nume cu valoarea pe care programul a specificat-o prin directiva *#define* corespunzătoare.

Atunci când definiți constante denumite în cadrul programelor, C++ nu restrânge aceste constante la valori numerice. Puteți folosi constante denumite pentru a reprezenta la fel de bine șiruri sau numere în virgulă mobilă. De exemplu, programul următor, *BookInfo.CPP*, folosește directiva *#define* pentru a defini trei constante care conțin informații despre această carte:

```
#include <iostream.h>

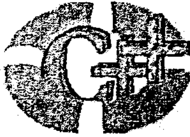
#define TITLE "Rescued by C++, Third Edition"
#define LESSON 17
#define PRICE 29.95

void main(void)
  {
    cout << "Book Title: " << TITLE << endl;
    cout << "Current Lesson: " << LESSON << endl;
    cout << "Price: $" << PRICE << endl;
  }
```

La compilarea și rularea programului *BookInfo.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> BookInfo <Enter>
Book Title: Rescued by C++, Third Edition
Current Lesson: 17
Price: $29.95
```

Utilizarea directivei #define pentru crearea constantelor denumite



Pentru a spori lizibilitatea programelor, ar trebui să înlocuiți valorile numerice care apar în cod cu constante denumite sugestive. Pentru a defini o constantă denumită, programele apelează la directiva preprocesor *#define*. Constantele denumite ar trebui plasate în partea de început a fișierului sursă. În plus, pentru a deosebi constantele de variabile, majoritatea programatorilor folosesc în numele de constante litere mari. De exemplu, directiva *#define* de mai jos creează o constantă numită *SECUNDE_PE_ORA*:

```
#define SECUNDE_PE_ORA 3600
```

La compilarea programului, preprocesorul de C++ va înlocui fiecare apariție a numelui *SECUNDE_PE_ORA* cu valoarea numerică 3600. Observați că definiția constantei nu se încheie cu punct și virgulă. Dacă ați fi introdus un punct și virgulă după 3600, preprocesorul de C++ ar înlocui apoi fiecare apariție a numelui *SECUNDE_PE_ORA* cu valoarea însoțită de punct și virgulă (3600;), generând probabil o eroare de sintaxă.

Utilizarea constantelor denumite pentru înlesnirea modificărilor în cod

Pe lângă sporirea lizibilității programelor, constantele denumite fac programele mai ușor de modificat. Spre exemplu, următorul fragment de cod conține mai multe instanțe ale numărului 50 (numărul de elevi dintr-o clasă):

```
#include <iostream.h>

void main(void)
  {
    int student;

    for (student = 0; student < 50; student++)
      get_test_score(student);

    for (student = 0; student < 50; student++)
      calculate_grade(student);

    for (student = 0; student < 50; student++)
      print_grade(student);
  }
```

Să presupunem acum că numărul de elevi dintr-o clasă crește la 55. În acest caz, va trebui să editați programul de mai sus și să înlocuiți fiecare apariție a numărului 50 cu 55. Ca o alternativă, programul următor folosește constanta denumită *CLASS_SIZE*:

```
#include <iostream.h>

#define CLASS_SIZE 50

void main(void)
  {
    int student;

    for (student = 0; student < CLASS_SIZE; student++)
      get_test_score(student);

    for (student = 0; student < CLASS_SIZE; student++)
      calculate_grade(student);

    for (student = 0; student < CLASS_SIZE; student++)
      print_grade(student);
  }
```

În acest exemplu, pentru modificarea mărimii unei clase, în întreg programul nu trebuie decât să schimbați linia care conține directiva *#define* ce definește constanta corespunzătoare:

```
#define CLASS_SIZE 55
```

Înlocuirea formulelor cu macrodefiniții

Atunci când programele efectuează diverse calcule, în cod apar de multe ori expresii complexe, ca cea prezentată aici:

```
result = (x*y-3) * (x*y-3) * (x*y-3);
```

În exemplul de mai sus, programul calculează cubul expresiei (x*y-3). Pentru a face programul mai ușor de citit și pentru a reduce șansele de a provoca o eroare prin tastarea eronată a unei expresii, puteți crea o macrodefiniție numită *CUBE* pe care programul să o folosească așa cum se arată în continuare:

```
result = CUBE(x*y-3);
```

Și în acest caz, programatorii obișnuiesc să folosească majuscule pentru numele de macrodefiniții, pentru a le putea deosebi de funcții.

Crearea unei macrodefiniţii se face prin intermediul directivei preprocesor *#define*. Instrucţiunea următoare, de pildă, creează macrodefiniţia *CUBE*:

```
#define CUBE(x)  ((x)*(x)*(x))
```

Aşa cum puteţi vedea, programul foloseşte instrucţiunea *#define* pentru a crea macrodefiniţia *CUBE* ce înmulţeşte parametrul *x* cu sine însuşi de două ori. Programul următor, *ShowCube.CPP*, utilizează macrodefiniţia *CUBE* pentru a afişa cubul fiecărui număr între 1 şi 10:

```
#include <iostream.h>

#define CUBE(x)  ((x)*(x)*(x))

void main(void)
  {
    for (int i = 1; i <= 10; i++)
      cout << i << " cubed is " << CUBE(i) << endl;
  }
```

La compilarea programului *ShowCube.CPP*, preprocesorul de C++ înlocuieşte fiecare apariţie a numelui *CUBE* cu macrodefiniţia corespunzătoare. Cu alte cuvinte, înlocuirile efectuate de preprocesor au ca rezultat următorul cod:

```
#include <iostream.h>

#define CUBE(x)  ((x)*(x)*(x))

void main(void)
  {
    for (int i = 1; i <= 10; i++)
      cout << i << " cubed is " << CUBE(i) << endl;
  }

#include <iostream.h>

#define CUBE(x)  ((x)*(x)*(x))

void main(void)
  {
    for (int i = 1; i <= 10; i++)
      cout << i << " cubed is " << ((i)*(i)*(i)) << endl;
  }
```

Remarcați că macrodefiniția prezentată a plasat parametrul *x* între paranteze sub forma *((x)*(x)*(x))*, prin contrast cu varianta *(x*x*x)*. Atunci când creați macrodefiniții, este recomandat să plasați parametrii între paranteze într-o manieră similară pentru a vă asigura că C++ va evalua expresiile așa cum doriți. După cum vă amintiți probabil din lecția 6, „Efectuarea de operații elementare", C++ apelează la prioritățile operatorilor pentru a determina ordinea de efectuare a operațiilor aritmetice. Să presupunem, de pildă, că un program folosește macrodefiniția *CUBE* pentru expresia *3+5-2*, ca mai jos:

```
result = CUBE(3+5-2);
```

Dacă macrodefiniția plasează parametrul între paranteze, preprocesorul va genera următoarea instrucțiune:

```
result = ((3+5-2) * (3+5-2) * (3+5-2));
```

Dacă, în schimb, macrodefiniția nu recurge la paranteze, preprocesorul va genera următoarea instrucțiune:

```
result = (3+5-2*3+5-2*3+5-2);
```

Dacă vă opriți un moment pentru a calcula fiecare expresie, veți vedea că rezultatele sunt diferite. Prin plasarea între paranteze a parametrilor unei macrodefiniții veți evita astfel de erori.

Diferențele dintre macrodefiniții și funcții

O macrodefiniție nu este o funcție. Atunci când un program folosește o funcție, în programul executabil se află o singură copie a instrucțiunilor acelei funcții. De fiecare dată când funcția este apelată, programul depune parametrii pe stivă și apoi efectuează saltul la codul funcției. După încheierea funcției, programul extrage valorile de pe stivă și sare înapoi la instrucțiunea imediat următoare apelului de funcție.

În cazul unei macrodefiniții, pe de altă parte, preprocesorul înlocuiește fiecare apariție din cod a numelui acesteia cu definiția corespunzătoare. De exemplu, dacă programul anterior ar fi folosit macrodefiniția *CUBE* în 100 de locuri diferite, preprocesorul ar fi generat în cod secvența de definiție de 100 de ori. Macrodefinițiile nu implică același efort ca și un apel de funcție (efortul de a depune și extrage parametri de pe stivă și efortul de salt către și de la codul funcției). Aceasta se întâmplă deoarece, în cazul macrodefinițiilor, preprocesorul plasează instrucțiunile corespunzătoare chiar acolo unde sunt solicitate. Cum preprocesorul înlocuiește, însă, fiecare nume al unei macrodefiniții cu codul corespunzător, macrodefinițiile sporesc dimensiunea programului executabil.

Utilizarea macrodefinițiilor este foarte flexibilă

Există o multitudine de feluri în care puteți utiliza macrodefinițiile în cadrul programelor. Rețineți, însă, că scopul utilizării acestor macrodefiniții este simplificarea codului și sporirea lizibilității programului. Următorul program, *MacDelay.CPP*, ilustrează flexibilitatea macrodefinițiilor. În plus, programul vă poate oferi o perspectivă mai bună asupra modului în care preprocesorul înlocuiește numele unei macrodefiniții cu instrucțiunile corespunzătoare:

```
#include <iostream.h>

#define delay(x) { \
          cout << "Delaying for   << x << endl; \
          for(long int i=0; i < x; i++) \
            \
          }

void main(void)
  {
    delay(100000L);

    delay(200000L);

    delay(300000L);
  }
```

În acest caz, deoarece definiția macro se întinde pe mai multe linii, fiecare linie care se continuă cu o alta este încheiată printr-un caracter backslash (\). Atunci când întâlnește numele macrodefiniției, preprocesorul îl înlocuiește cu întreaga serie de instrucțiuni care apar în definiția macro.

Ce trebuie să știți

Macrodefinițiile și constantele denumite au rolul de a spori lizibilitatea programelor și înlesni programarea. Lecția de față a studiat modul de creare și utilizare în cod a constantelor denumite și a macrodefinițiilor. În lecția 18, „Păstrarea valorilor multiple în cadrul vectorilor", veți învăța să plasați mai multe valori de același tip în cadrul unui *vector*. De pildă, un program ar putea reține 100 de note la examen sau 50 de prețuri de acțiuni. Prin intermediul vectorilor, păstrarea și utilizarea unor astfel de valori este foarte facilă. Dar înainte de a trece la lecția 18, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Macrodefinițiile și constantele denumite fac programele mai ușor de citit prin înlocuirea formulelor complexe și a valorilor numerice cu nume sugestive.

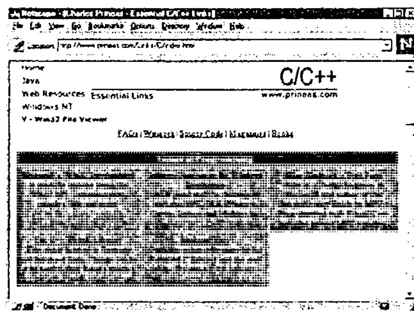
- ☑ Prin înlocuirea valorilor numerice dintr-un program cu constante denumite reduceți numărul modificărilor pe care va trebui să le efectuați asupra programului mai târziu în cazul în care valoarea constantei s-ar schimba.

- ☑ În timpul compilării, compilatorul de C++ folosește un program special, numit preprocesor, pentru a înlocui fiecare constantă denumită și macrodefiniție cu valoarea corespunzătoare.

- ☑ Macrodefinițiile se execută mai repede decât funcțiile, dar au ca efect creșterea dimensiunii programului executabil.

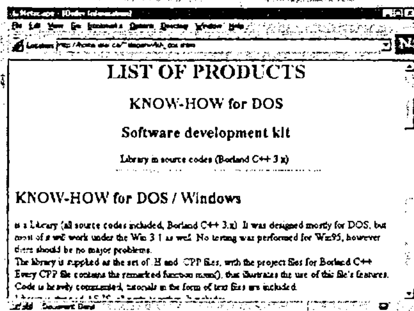
- ☑ Dacă o macrodefiniție se întinde pe mai multe linii, plasați caracterul backslash la sfârșitul fiecărei linii corespunzătoare pentru a informa preprocesorul că definiția continuă și pe linia următoare.

LEGĂTURI C++



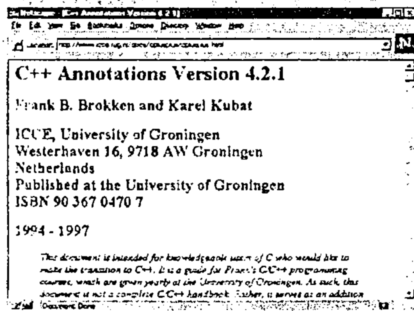
http://www.prineas.com/Links/C/index.html

SECRETE DOS



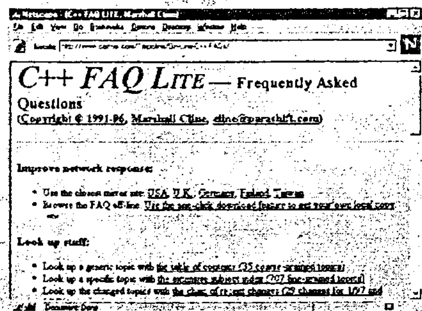
http://home.istar.ca/~stepanv/kb_dos.shtml

INFORMAȚII DESPRE C++ 4.2.1.



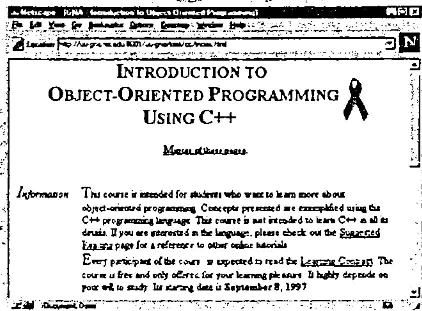
http://www.icce.rug.nl/docs/cplusplus.html

CELE MAI FRECVENTE ÎNTREBĂRI



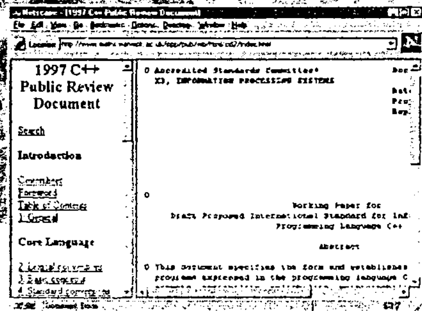
http://www.cerfnet.com/~mpcline/On-Line-C++FAQs/

PROGRAMAREA ORIENTATĂ SPRE OBIECT



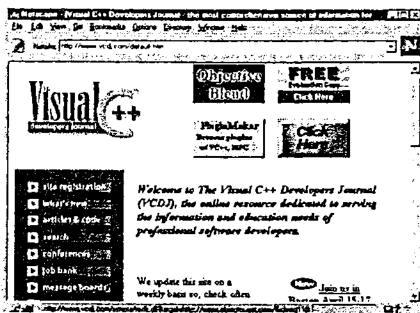
http://uu-gna.mit.edu:8001/uu-gna/text/cc/index.html

1997 C++ PUBLIC REVIEW



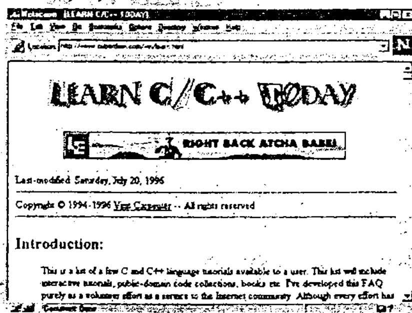
http://www.maths.warwick.ac.uk/cpp/pub/wp/html/cd2/index.html

Revista Programatorilor Visual C++



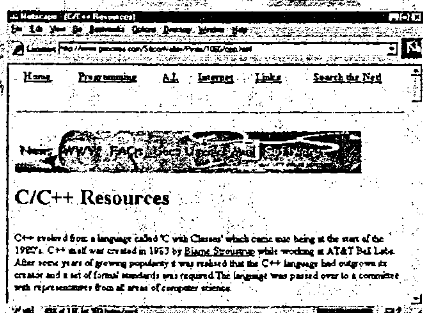
http://www.vcdj.com/default.htm

Învățați C/C++



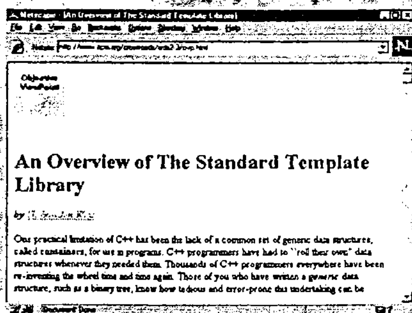
http://www.cyberdiem.com/vin/learn.html

Resurse C/C++



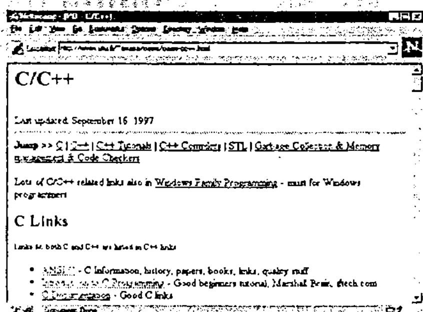
http://www.geocities.com/SiliconValley/Pines/1060/cpp.html

Biblioteca de șabloane standard



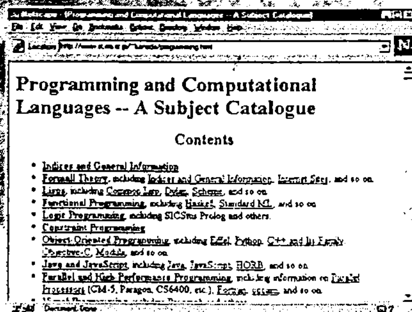
http://www.acm.org/crossroads/xrds2-3/opv.html

PO-C/C++



http://www.utu.fi/~sisasa/oasis-cc++.html

Limbaje matematice



http://www.st.rim.or.jp/~kanada/programming.html

PARTEA a III-a

Păstrarea informațiilor cu ajutorul vectorilor și al structurilor

Așa cum ați învățat, o variabilă poate reține o valoare de un tip anume. În programele C++ pe care le-ați creat pe parcursul primelor două părți ale acestei cărți, fiecare variabilă putea reține o singură valoare. Odată cu creșterea complexității programelor, vor fi cazuri în care veți fi nevoit să lucrați simultan cu mai multe valori. Spre exemplu, un program ar putea lucra cu 100 de note la examen, cu 30 de cotații de acțiuni sau cu numele și adresele tuturor celor 5000 de angajați ai unei firme. În cadrul acestei părți veți învăța să folosiți diferite tipuri de date din C++ în scopul păstrării simultane într-o variabilă a mai multor valori. După cum veți vedea, utilizarea unei singure variabile pentru reținerea mai multor valori este foarte utilă. Lecțiile cuprinse în această parte sunt următoarele:

Lecția 18

Păstrarea valorilor multiple în cadrul vectorilor

Pe durata rulării unui program, informațiile acestuia sunt păstrate în variabile. Până acum, variabilele din programele create puteau reține o singură valoare la un moment dat. În multe cazuri, însă, programele au nevoie să rețină mai multe valori, precum 50 de note la examen, 100 de titluri de cărți sau 1000 de nume de fișiere. Pentru a păstra mai multe valori, programele folosesc o structură de date specială care se numește *vector*. Pe scurt, un vector este pur și simplu o variabilă care poate reține mai multe valori de același tip, cum ar fi 10 valori de tip *int*. Pentru declararea unui vector, programele trebuie să specifice tipul și numele vectorului, precum și numărul de elemente pe care acesta le va conține. Lecția de față se oprește asupra modului în care programele declară un vector și ulterior plasează și extrag informațiile în cadrul acestuia. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Un *vector* este o structură de date care permite unei singure variabile să păstreze mai multe valori de același tip.
- La declararea unui vector trebuie specificat tipul valorilor pe care acesta le va reține, precum și numărul de elemente ce urmează a fi conținute (numite *elemente ale vectorului*).
- Toate elementele dintr-un vector trebuie să aibă același tip, precum *int*, *float* sau *char*.
- Pentru plasarea unei valori într-un vector trebuie să specificați numărul elementului din vector care doriți să rețină valoarea respectivă. De exemplu, primul element al unui vector este elementul 0, al doilea este elementul 1 și așa mai departe.
- Pentru accesarea unei valori aflate într-un vector, programele trebuie să precizeze numele vectorului și numărul elementului, plasându-l pe acesta din urmă între paranteze pătrate, ca de pildă *note[3]*.
- La declararea unui vector, programul poate utiliza operatorul de atribuire pentru a inițializa elementele vectorului.
- Programele pot transmite către funcții variabile vector așa cum ar transmite orice parametru.

Programele C++ utilizează intens vectori. În lecția 19, „Despre șirurile de caractere", veți lucra cu șiruri de caractere (așa cum sunt titlul unei cărți, numele unui fișier etc.) și vectori de caractere.

Declararea unei variabile vector

Un *vector* este o variabilă care poate păstra una sau mai multe valori de același tip. Asemeni variabilelor folosite până acum în programe, un vector trebuie să aibă un tip (precum *int*, *char* sau *float*) și un nume unic. În plus, trebuie specificat numărul de valori pe care le va conține vectorul. Toate valorile păstrate într-un vector trebuie să aibă același tip. Cu alte cuvinte, un program nu poate plasa în același vector valori de tip *float*, *char* și *long*. Declarația următoare creează un vector numit *note_examen* care poate reține 100 de note întregi:

```
int note_examen[100];
```

Tipul vectorului

Dimensiunea vectorului

Atunci când întâlnește declarația vectorului, compilatorul de C++ alocă memoria necesară pentru păstrarea a 100 de valori de tipul *int*. Valorile pe care programul le păstrează într-un vector reprezintă *elementele vectorului*.

Vectorii păstrează mai multe valori având același tip



Pe măsură ce programele cresc în complexitate, ele vor ajunge să lucreze cu mai multe valori de un același tip. Spre exemplu, un program ar putea reține prețurile a 50 de componente, vârstele a 100 de angajați sau cotațiile pentru 25 de acțiuni. În loc să vă oblige la folosirea în program a 50, 100 sau 25 de variabile cu nume unice, C++ vă permite definirea în program a unei singure variabile, un vector, care va putea reține mai multe valori înrudite.

Pentru declararea unui vector trebuie să specificați un tip al vectorului, un nume unic și numărul de elemente pe care le va conține vectorul. Spre exemplu, instrucțiunile următoare declară trei vectori diferiți:

```
float pret_componenta[50];
int varsta_angajat[100];
float cotatie_actiune[25];
```

Accesarea elementelor vectorului

După cum ați aflat, un vector permite programului să păstreze mai multe valori într-o aceeași variabilă. Pentru a accesa diferitele valori pe care programul le poate reține într-un vector, veți folosi o *valoare de index* care indică elementul vizat. Spre exemplu, pentru a accesa primul element din vectorul *note_examen* veți preciza valoarea de index 0 (*note_examen[0]*). Pentru accesarea celui de-al doilea element specificați valoarea de index 1 (*note_examen[1]*). Analog, pentru accesarea celei de a treia valori veți preciza valoarea de index 2 (*note_examen[2]*). Așa cum este ilustrat în figura 18.1, programele C++ numerotează întotdeauna primul element al unui vector cu 0 și ultimul element din vector cu o valoare egală cu dimensiunea vectorului minus unu.

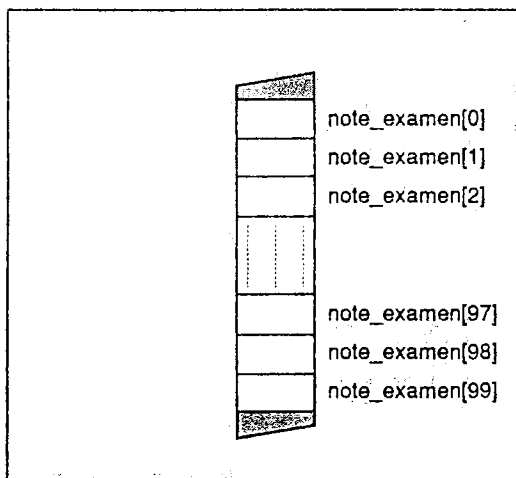


Figura 18.1 Modul în care C++ numerotează elementele unui vector.

Este important să reţineţi că C++ numerotează întotdeauna primul element din vector cu 0 şi ultimul element cu dimensiunea vectorului minus 1. Programul următor, *Array.CPP*, creează un vector numit *values* care poate păstra cinci valori întregi. Programul atribuie apoi elementelor valorile 100, 200, 300, 400 şi 500:

```
#include <iostream.h>

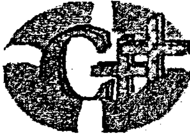
void main(void)
  {
    int values[5]; // Declaratia vectorului

    values[0] = 100;
    values[1] = 200;
    values[2] = 300;
    values[3] = 400;
    values[4] = 500;

    cout << "The array contains the following values"
      << endl;
    cout << values[0] << ' ' << values[1] << ' '
      << values[2] << ' ' << values[3] << ' '
      << values[4] << endl;
  }
```

După cum vedeți, programul atribuie prima valoare elementului 0 (*values[0]*). De asemenea, ultima valoare este atribuită de program elementului 4 (dimensiunea vectorului (5) minus 1).

Utilizarea unei valori de index pentru accesarea elementelor din vectori



Un vector permite programelor plasarea mai multor valori de același tip în cadrul aceleiași variabile. Pentru a accesa valori anume din vector, programele folosesc o valoare de index. Pe scurt, o valoare de index precizează elementul dorit din vector. Toți vectorii din C++ încep cu elementul 0. Instrucțiunea următoare, de pildă, atribuie valoarea 100 primului element al vectorului *note*:

```
note[0] = 100;
```

La declararea unui vector, programul specifică numărul de elemente pe care le va putea reține vectorul respectiv. De exemplu, instrucțiunea următoare declară un vector ce poate reține 100 de valori de tip *int*:

```
int note[100];
```

În acest caz, elementele vectorului încep cu *note[0]* și se termină cu *note[99]*.

Utilizarea unei variabile index

În cazul utilizării vectorilor în programe, o operație uzuală este folosirea unei variabile index pentru a accesa elementele vectorului. De exemplu, presupunând că variabila index *i* conține valoarea 3, instrucțiunea următoare atribuie valoarea 400 elementului *values[3]*:

```
values[i] = 400;
```

Programul următor, *ShowArra.CPP*, utilizează variabila index *i* în cadrul unei bucle *for* pentru a afișa elementele unui vector. Bucla *for* inițializează *i* cu 0, astfel încât să acceseze elementul *value[0]*, și se încheie atunci când *i* este mai mare decât 4 (ultimul element al vectorului):

```
#include <iostream.h>

void main(void)
  {
```

```
    int values[5]; // Declaratia vectorului
    int i;
    values[0] = 100;
    values[1] = 200;
    values[2] = 300;
    values[3] = 400;
    values[4] = 500;
    cout << "The array contains the following values"
      << endl;
    for (i = 0; i < 5; i++)
       cout << values[i] << ' ';
}
```

De fiecare dată când bucla *for* incrementează variabila *i*, programul ajunge să acceseze următorul element din vector. Experimentați cu programul *ShowArra.CPP*, înlocuind bucla *for* cu următoarea:

```
for (i = 4; i >= 0; i--)
   cout << values[i] << ' ';
```

În acest caz, programul va afişa elementele vectorului începând cu ultimul şi terminând cu primul.

Inițializarea unui vector la declarare

După cum ați văzut, C++ permite programelor inițializarea variabilelor odată cu declararea acestora. Acelaşi lucru este posibil şi în cazul vectorilor. La declararea unui vector puteți specifica valorile inițiale prin plasarea acestora între acolade după un semn egal. De exemplu, instrucțiunea următoare inițializează vectorul *values*:

```
int values[5] = { 100, 200, 300, 400, 500 };
```

În mod similar, declarația următoare inițializează un vector de valori în virgulă mobilă:

```
float salaries[3] = { 25000.00, 35000.00, 50000.00 };
```

Dacă nu precizați valori inițiale pentru elementele vectorului, majoritatea compilatoarelor de C++ inițializează elementele cu valoarea 0. De exemplu, declarația următoare inițializează primele trei elemente dintr-un vector cu cinci elemente:

```
int values[5] = { 100, 200, 300 };
```

În exemplul anterior, instrucțiunea nu a inițializat elementele *values[3]* și *values[4]*. În funcție de compilator, este posibil ca acestor elemente să le fie atribuită valoarea 0. Dacă nu specificați dimensiunea unui vector pe care-l inițializați la declarare, C++ va aloca suficientă memorie pentru a reține numărul de elemente corespunzător valorilor precizate. Spre exemplu, declarația următoare creează un vector ce poate reține patru valori întregi:

```
int numbers[] = { 1, 2, 3, 4 };
```

Transmiterea de vectori către funcții

Programele pot transmite vectori către funcții asemeni oricăror altor variabile. Funcțiile ar putea apoi să inițializeze vectorul, să adune valorile conținute sau să afișeze pe ecran elementele din vector. Atunci când transmiteți un vector unei funcții trebuie să specificați tipul acelui vector. Nu este nevoie să precizați și dimensiunea vectorului. Veți transmite, în schimb, un alt parametru, așa cum ar fi variabila *numar_de_elemente*, care va indica numărul de elemente din vector, după cum se vede aici:

```
void o_functie(int vector[], int numar_de_elemente);
```

Programul următor, *ArrayFun.CPP*, transmite vectori funcției *show_array*, iar aceasta folosește o buclă *for* pentru a afișa valorile din vector:

```
#include <iostream.h>

void show array(int array[], int_number_of_elements)
  {
    int i;

    for (i = 0; i < number_of_elements; i++)
      cout << array[i] << ' ';

    cout << endl;
  }

void main(void)
  {
    int little_numbers[5] = { 1, 2, 3, 4, 5 };
```

```
    int big_numbers[3] = { 1000, 2000, 3000 };

    show_array(little_numbers, 5);
    show_array(big_numbers, 3);
  }
```

După cum puteți observa, programul transmite funcției un vector prin numele acestuia, specificând totodată un parametru care indică funcției numărul de elemente aflate în vector, ca mai jos:

```
show_array(little_numbers, 5);
```

Programul următor, *GetArray.CPP*, utilizează funcția *get_values* pentru a fixa cele trei valori ale vectorului *numbers*:

```
#include <iostream.h>

void get values(int array[], int number_of_elements)
  {
    int i;

    for (i = 0; i < number_of_elements; i++)
    {
      cout << "Enter value " << i << ": ";
      cin >> array[i];
    }
  }

void main(void)
  {
    int numbers[3];

    get_values(numbers, 3);

    cout << "The array values are as follows" << endl;

    for (int i = 0; i < 3; i++)
      cout << numbers[i] << endl;
  }
```

După cum se vede, programul transmite funcției vectorul prin numele său. Funcția fixează apoi valorile pentru elementele vectorului. În lecția 11, „Modificarea valorilor parametrilor", ați văzut că o funcție nu poate modifica un parametru decât dacă acesta este transmis

de către program funcţiei prin adresa sa. Aşa cum se observă aici, însă, funcţia *get_values* modifică parametrul *numbers*, care este un vector. După cum veţi afla din lecţia 22, „Despre pointeri", C++ transmite de fapt vectorii către funcţii sub forma unui pointer (o adresă de memorie). Din acest motiv, funcţia poate modifica în orice fel elementele vectorului.

Ce trebuie să ştiţi

În lecţia de faţă aţi învăţat că programele pot păstra mai multe valori de acelaşi tip în cadrul unui vector. Programele C++ folosesc în mod frecvent vectori. În lecţia 19, „Despre şirurile de caractere", veţi vedea că programele C++ utilizează vectorii pentru reţinerea şirurilor de caractere. Dar înainte de a trece la lecţia 19, asiguraţi-vă că aţi reţinut următoarele aspecte importante:

- ☑ Un vector este o variabilă care poate păstra una sau mai multe valori de acelaşi tip.
- ☑ Pentru a declara un vector trebuie să specificaţi un tip, numele vectorului şi numărul de valori pe care le va păstra acesta.
- ☑ Valorile pe care programul le păstrează într-un vector constituie elementele vectorului.
- ☑ Primul element din vector este elementul zero (*vector[0]*); ultimul element din vector are indexul cu unu mai mic decât dimensiunea vectorului.
- ☑ Programele utilizează, de regulă, variabile index pentru accesarea elementelor din vectori.
- ☑ Atunci când primeşte ca parametru un vector, o funcţie trebuie să precizeze tipul şi numele vectorului, dar nu şi dimensiunea acestuia.
- ☑ Atunci când transmite un parametru unei funcţii, programul transmite, de obicei, şi un parametru care indică funcţiei numărul de elemente pe care le conţine vectorul.
- ☑ Deoarece C++ transmite vectorii către funcţii prin adresa acestora, funcţiile pot modifica valorile aflate în vectori.

Lecția 19
Despre șirurile de caractere

În programele C++, șirurile de caractere rețin informații care pot fi nume de fișiere, titluri de cărți, nume de angajați sau alte combinații de caractere. Majoritatea programelor C++ utilizează intens șiruri de caractere. Așa cum veți vedea în lecția de față, C++ păstrează șirurile de caractere în cadrul vectorilor de tip *char*. Veți învăța să rețineți și să manipulați șiruri de caractere și să profitați de funcțiile din biblioteca de execuție care au ca scop prelucrarea șirurilor de caractere. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru declararea unui șir de caractere trebuie să declarați un vector de tip *char*.
- Pentru a fixa valoarea unui șir de caractere, programul atribuie caractere elementelor din vectorul șir de caractere.
- Programele C++ folosesc caracterul NULL (cod ASCII 0) pentru a marca ultimul caracter al unui șir.
- C++ permite programelor inițializarea șirurilor de caractere în momentul declarării variabilelor șir de caractere.
- Programele pot transmite șiruri de caractere către funcții așa cum ar transmite orice alt vector.
- Majoritatea bibliotecilor de execuție C++ oferă o mulțime de funcții care pot prelucra șiruri de caractere.

Programele C++ rețin șirurile de caractere sub forma unui vector de tip *char*. Majoritatea programelor C++ utilizează intensiv șiruri de caractere. Experimentați cu fiecare dintre programele prezentate în lecția de față pentru a vă asigura că v-ați familiarizat cu șirurile de caractere. După cum veți vedea, utilizarea șirurilor de caractere este similară cu utilizarea vectorilor de alte tipuri despre care am discutat în lecția 18, „Păstrarea valorilor multiple în cadrul vectorilor".

Declararea unui șir de caractere în cadrul programului

Programele C++ folosesc frecvent șiruri de caractere pentru a reține nume de utilizatori, nume de fișiere sau alte informații de tip literal. Pentru a declara un șir de caractere în cadrul unui program, este suficient să declarați un vector de tip *char* suficient de mare pentru a putea păstra caracterele dorite. De exemplu, declarația următoare creează o variabilă șir de caractere numită *nume_fisier*, care poate reține 64 de caractere (nu uitați că unul dintre aceste 64 de caractere este și caracterul NULL pe care-l folosiți pentru a marca sfârșitul șirului):

```
char nume_fisier[64];
```

Așa cum ilustrează figura 19.1, această declarație creează un vector ce conține elementele de la *nume_fisier[0]* până la *nume_fisier[63]*.

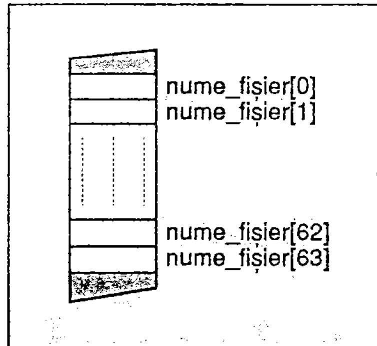


Figura 19.1 *C++ tratează șirurile de caractere ca vectori de tip **char**.*

Principala diferență dintre șirurile de caractere și celelalte tipuri de vectori este modul în care C++ indică ultimul element al vectorului. Veți vedea că programele C++ marchează sfârșitul unui șir de caractere cu ajutorul caracterului NULL, reprezentat în C++ de caracterul special '\0'. La stabilirea valorii caracterelor dintr-un șir trebuie să plasați după ultimul caracter al șirului caracterul NULL ('\0'). De exemplu, programul următor, *Alphabet.CPP*, atribuie variabilei *alphabet* mulțimea literelor de la A la Z prin intermediul unei bucle *for*. Programul atașează apoi variabilei caracterul NULL și afișează variabila utilizând *cout*:

```
#include <iostream.h>

void main(void)
  {
    char alphabet[27]; // 26 de litere si caracterul NULL
    char letter;
    int index;

    for (letter = 'A' index = 0; letter <= 'Z';
      letter++, index++)
       alphabet[index] = letter;
    alphabet[index] = NULL;
    cout << "The letters are " << alphabet;
  }
```

Precum vedeţi, programul plasează în şir caracterul NULL pentru a marca ultimul caracter al şirului:

```
alphabet[index] = NULL;
```

Atunci când fluxul de ieşire *cout* afişează şirul de caractere, acesta va afişa caracterele din şir unul câte unul, până la întâlnirea caracterului NULL. Pe scurt, caracterul NULL indică programului (sau funcţiilor din biblioteca de execuţie) poziţia ultimului caracter din şir.

Priviţi mai atent bucla *for* care apare în programul de mai sus. După cum puteţi vedea, bucla iniţializează şi incrementează două variabile (*letter* şi *index*). Atunci când o buclă *for* iniţializează sau incrementează mai multe variabile, aceste operaţii se separă prin operatorul *virgulă* din C++:

```
for (letter = 'A'  index = 0; letter <= 'Z'; letter++,
  index++)
```

C++ ataşează automat caracterul NULL constantelor şir de caractere

Toate programele pe care le-aţi creat pe parcursul acestei cărţi au utilizat constante şir de caractere plasate între ghilimele, ca mai jos:

```
"Aceasta este o constanta sir de caractere"
```

Atunci când creaţi o constantă şir de caractere, compilatorul de C++ ataşează automat caracterul NULL, aşa cum înfăţişează figura 19.2.

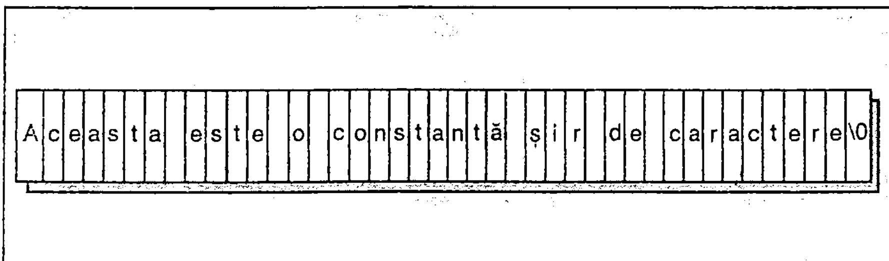


Figura 19.2 Compilatorul de C++ ataşează automat caracterul NULL unei constante şir de caractere.

Atunci când un program afişează constantele şir de caractere prin intermediul fluxului de ieşire *cout*, acesta ţine cont de caracterul NULL pe care compilatorul l-a ataşat şirului pentru a marca ultimul caracter de afişat.

Utilizarea caracterului NULL



Un șir de caractere este un vector de caractere încheiat cu caracterul NULL ('\0'). Declararea unui șir de caractere se face prin declararea unui vector de tip *char*. Atunci când un program stabilește ulterior caracterele unui șir, programul este responsabil pentru atașarea caracterului NULL care marchează sfârșitul șirului. La utilizarea constantelor șir de caractere plasate între ghilimele, compilatorul de C++ atașează automat caracterul NULL. Majoritatea funcțiilor din C++ folosesc prezența caracterului NULL pentru a determina ultimul caracter al unui șir.

Programul următor, *LoopNull.CPP*, modifică puțin programul anterior pentru a utiliza o buclă *for* care afișează conținutul șirului:

```
#include <iostream.h>

void main(void)
  {
    char alphabet[27]; // 26 de litere si caracterul NULL
    char letter;
    int index;

    for (letter = 'A', index = 0; letter <= 'Z'; letter++,
      index++)
       alphabet[index] = letter;

    alphabet[index] = '/0';

    for (index = 0; alphabet[index] != '\0'; index++)
       cout << alphabet[index];
    cout << end;
  }
```

După cum puteți vedea, bucla *for* parcurge șirul caracter cu caracter. În cazul în care caracterul curent nu este NULL (care ar fi indicat ultimul caracter din șir), bucla va afișa acest caracter, după care va incrementa indexul, iar procesul va continua.

Diferența dintre 'A' și "A"

Pe măsură ce veți inspecta diferite programe C++, veți întâlni unele caractere încadrate de apostrofuri (cum este 'A') și alte caractere încadrate de ghilimele (cum este "A"). Un caracter încadrat de apostrofuri este o *constantă caracter*. Compilatorul C++ alocă un singur octet de memorie pentru a reține o constantă caracter. Un caracter între ghilimele, însă, reprezintă o *constantă șir de caractere*, conținând caracterul în cauză și caracterul

NULL (pe care îl atașează compilatorul). Astfel, compilatorul va aloca doi octeți de memorie pentru a reține o constantă șir de caractere. Figura 19.3 ilustrează modul în care compilatorul de C++ reține constanta caracter 'A' și constanta șir de caractere "A".

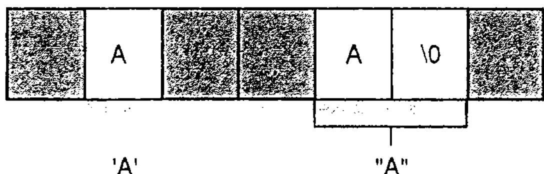


Figura 19.3 *Modul în care compilatorul C++ reține constanta caracter 'A' și, respectiv, constanta șir de caractere "A".*

Inițializarea unui șir de caractere

Așa cum ați văzut în lecția 18, compilatorul de C++ vă permite inițializarea vectorilor la declarare. Șirurile de caractere din C++ nu fac nici ele excepție. Pentru a inițializa un șir de caractere la declararea sa, precizați caracterele șirului încadrate de ghilimele, ca mai jos:

```
char title[64] = "Rescued by C++"
```

Dacă numărul de caractere pe care le atribuiți șirului este mai mic decât dimensiunea vectorului, majoritatea compilatoarelor de C++ vor atribui restului de caractere valoarea NULL. Ca și în cazul vectorilor, dacă nu precizați dimensiunea unui vector șir de caractere pe care îl inițializați la momentul declarării, compilatorul de C++ va aloca memoria necesară pentru a reține literele specificate și caracterul NULL. Spre exemplu, instrucțiunea următoare creează un șir de caractere numit *title* care conține literele specificate și un caracter NULL:

```
char title[64] = "Rescued by C++"
```

Programul următor, *Init_Str.CPP*, inițializează un șir de caractere la momentul declarării sale:

```
#include <iostream.h>

void main(void)
  {
    char title[64] = "Rescued by C++";
    char lesson[64] = "Understanding Character Strings";

    cout << "Book: " << title << endl;
    cout << "Lesson: " << lesson << endl;
  }
```

Mai multe dintre programele care vor fi prezentate în continuarea acestei cărți vor apela la această metodă de inițializare a șirurilor. Acordați-vă câteva minute pentru a experimenta cu programul *Init_Str.CPP*, modificând caracterele pe care programul le atribuie celor două șiruri.

Transmiterea de șiruri către funcții

Transmiterea unui șir de caractere către o funcție este foarte asemănătoare cu transmiterea ca parametru a unui vector oarecare. Este suficient să specificați în cadrul funcției tipul vectorului (*char*) și cele două paranteze pătrate. Nu trebuie să precizați dimensiunea șirului. De pildă, programul următor, *Show_Str.CPP*, utilizează funcția *show_string* pentru a afișa pe ecran un șir de caractere:

```
#include <iostream.h>

void show_string(char string[])
  {
    cout << string << endl;
  }

void main(void)
  {
    show_string("Hello, C++!");
    show_string("I've been Rescued by C++");
  }
```

După cum se vede, funcția *show_string* tratează parametrul șir de caractere ca pe un vector, așa cum se vede aici:

```
void show_string(char string[])
```

Deoarece sfârșitul șirului este indicat de caracterul NULL, funcția nu mai necesită un parametru care să specifice numărul de elemente din vector. Ea va putea identifica ultimul element căutând în vector caracterul NULL. Spre exemplu, programul următor, *ByChar.CPP*, transmite funcției *display_to_NULL* un șir de caractere, iar funcția afișează literele șirului una câte una, până la întâlnirea caracterului NULL:

```
#include <iostream.h>

void display_to_NULL(char string[])
  {
      for (int i = 0; string[i] != '\0'; i++)
```

```
            cout << string[i];
    }

void main(void)
    {
        display_to_NULL("Rescued by C++");
    }
```

Așa cum ați aflat, funcțiile C++ folosesc deseori prezența caracterului NULL pentru a determina sfârșitul unui șir.

Programul care urmează, *Str_Len.CPP*, creează o funcție numită *string_length* care caută într-un șir caracterul NULL pentru a determina numărul de caractere aflate în acel șir. Funcția folosește apoi o instrucțiune *return* pentru a întoarce apelantului lungimea șirului. Programul *Str_Len.CPP* transmite acestei funcții mai multe șiruri de caractere diferite, afișând pe ecran lungimea fiecărui șir:

```
#include <iostream.h>

int string_length(char string[])
    {
      int i;

      for (i = 0; string[i] != '\0'; i++); // Nu facem altceva
                                           // decat sa trecem
                                           // la urmatorul
                                           // caracter

      return(i); // Lungimea sirului
    }

void main(void)
    {
      char title[] = "Rescued by C++";
      char lesson[] = "Understanding Character Strings";

      cout << "The string " << title << " contains " <<
        string_length(title) << " characters" << endl;

      cout << "The string " << lesson << " contains " <<
        string_length(lesson) << " characters" << endl;
    }
```

După cum puteți vedea, funcția începe cu primul caracter al șirului (elementul 0) și parcurge apoi fiecare element până când întâlnește valoarea NULL. Pe măsură ce veți examina mai multe programe C++, veți întâlni multe funcții care caută în mod similar caracterul NULL dintr-un șir de caractere.

Profitați de faptul că NULL are codul ASCII 0

Precum am menționat, caracterul NULL are codul ASCII 0. În lecția 8, „Învățați programul să ia decizii", ați văzut că C++ folosește valoarea 0 pentru a reprezenta o valoare de fals. Prin urmare, cum caracterul NULL este egal cu 0, puteți simplifica multe dintre operațiile buclelor din programe. De exemplu, multe funcții parcurg șirurile de caractere element cu element, în căutarea valorii NULL. Bucla *for* următoare ilustrează o posibilă formă de căutare a caracterului NULL într-un șir:

```
for (index = 0; string[index] != NULL; index++)
```

Deoarece caracterul NULL are valoarea 0, multe programe simplifică forma buclelor care caută valoarea NULL în felul următor:

```
for (index = 0; string[index]; index++)
```

În acest exemplu, bucla continuă atât timp cât caracterul curent din *string[index]* este diferit de NULL (0 sau fals).

Utilizarea funcțiilor de manipulare a șirurilor din biblioteca de execuție

În lecția 12, „Utilizarea bibliotecilor de execuție", ați învățat că majoritatea compilatoarelor de C++ pun la dispoziție o foarte utilă colecție de funcții, numită biblioteca de execuție. Dacă inspectați această bibliotecă de execuție, veți vedea că ea conține multe funcții care au ca scop manipularea șirurilor de caractere. De exemplu, funcția *strupr* transformă literele unui șir în majuscule. De asemenea, funcția *strlen* întoarce numărul de caractere dintr-un șir. Cele mai multe biblioteci de execuție oferă chiar și funcții ce vă permit căutarea anumitor caractere într-un șir. Programul următor, *StrUpr.CPP*, ilustrează, de pildă, folosirea funcțiilor *strupr* și *strlwr* din biblioteca de execuție:

```
#include <iostream.h>
#include <string.h> // Contine prototipuri de functii

void main(void)
  {
    char title[] = "Rescued by C++";
    char lesson[] = "Understanding Character Strings";

    cout << "Uppercase: " << strupr(title) << endl;
    cout << "Lowercase: " << strlwr(lesson) << endl;
  }
```

Prin utilizarea funcţiilor de manipulare a şirurilor din cadrul bibliotecii de execuţie puteţi reduce considerabil efortul de programare. Acordaţi-vă puţin timp pentru a tipări o copie a fişierului de antet *string.h* în scopul identificării funcţiilor de manipulare a şirurilor pe care le oferă biblioteca de execuţie a compilatorului dumneavoastră.

Trebuie să respectaţi regulile



Aşa cum aţi văzut, cele mai multe funcţii care manipulează şiruri se bazează pe caracterul NULL pentru a determina ultimul caracter al unui şir. Atunci când atribuiţi valori şirurilor, asiguraţi-vă că programul ataşează şi caracterul NULL ca ultim caracter al şirului. Dacă programele dumneavoastră nu vor folosi cu consecvenţă valoarea NULL, funcţiile care se bazează pe prezenţa caracterului NULL vor eşua.

Ce trebuie să ştiţi

Majoritatea programelor C++ folosesc intens şiruri de caractere. În lecţia de faţă aţi învăţat să lucraţi cu şiruri în cadrul programelor. În lecţia 20, „Păstrarea informaţiilor înrudite în cadrul structurilor", veţi afla cum puteţi păstra informaţii de tipuri diferite într-o variabilă structură din C++. Prin intermediul structurilor puteţi păstra reunite într-o singură variabilă informaţii precum numele, vârsta, salariul şi numărul de telefon al unui angajat. Dar înainte de a trece la lecţia 20, asiguraţi-vă că aţi reţinut următoarele aspecte importante:

☑ Un şir de caractere este un vector de caractere pe care programele îl încheie cu caracterul ASCII 0 (NULL).

☑ Crearea unui şir de caractere într-un program se face prin declararea unui vector de tip *char*.

- ☑ Programul este cel responsabil pentru plasarea caracterului NULL după ultimul caracter din șir.

- ☑ Atunci când programul folosește constante șir de caractere încadrate de ghilimele, compilatorul de C++ atașează automat caracterul NULL.

- ☑ C++ vă permite inițializarea șirurilor la momentul declarării prin specificarea caracterelor între ghilimele.

- ☑ Majoritatea compilatoarelor de C++ oferă în cadrul bibliotecii de execuție o mulțime cuprinzătoare de funcții pentru manipularea șirurilor.

Lecția 20

Păstrarea informațiilor înrudite în cadrul structurilor

În lecția 18, „Păstrarea valorilor multiple în cadrul vectorilor", ați văzut că C++ vă permite plasarea informațiilor de același tip în cadrul vectorilor. Așa cum ați remarcat, plasarea împreună a unor valori înrudite în vectori este foarte convenabilă. Pe măsură ce programele dumneavoastră sporesc în complexitate, ați putea avea nevoie să plasați la un loc informații care nu au același tip. Spre exemplu, să presupunem că un program lucrează cu date despre angajați. Programul trebuie să rețină pentru fiecare angajat numele, vârsta, salariul, adresa, numărul de ordine, numărul de funcție și așa mai departe. Pentru a reține aceste informații, programul va avea nevoie de variabile de tip *char*, *int*, *float* și de șiruri de caractere.

Atunci când un program trebuie să păstreze informații înrudite ale căror tipuri diferă, puteți apela la o *structură*. După cum veți vedea, o structură este o variabilă care grupează elemente de informație înrudite, numite *membri*, ale căror tipuri pot fi diferite. Printr-o astfel de grupare a datelor într-o singură variabilă, programele devin mai simple, reducându-se numărul de variabile de gestionat, de transmis către funcții, etc. Lecția de față studiază modul în care programele pot crea și utiliza structuri. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Structurile permit programelor să grupeze într-o singură variabilă date înrudite, dar ale căror tipuri pot diferi.
- O structură este alcătuită dintr-unul sau mai multe elemente de date, numite *membri*.
- Pentru a defini o structură într-un program trebuie să specificați numele structurii și membrii acesteia.
- Fiecare membru al unei structuri are un tip anume, precum *char*, *int* sau *float*, iar fiecare nume de membru trebuie să fie unic în interiorul structurii.
- După definirea unei structuri, programul poate declara variabile ce au ca tip acea structură.
- Pentru a putea modifica membrii unei structuri în cadrul unei funcții, programele trebuie să transmită funcției respectiva structură prin adresa ei.

În partea a 4-a a acestei cărți veți studia programarea orientată spre obiect în C++. O bună înțelegere și utilizare a structurilor vor face ca folosirea claselor C++ să fie mult mai facilă. Experimentați cu fiecare program pe care îl prezintă această lecție și asigurați-vă că sunteți familiar cu operațiile de plasare și extragere a valorilor membrilor unei structuri.

Declararea unei structuri

O structură definește un *șablon* pe care programele îl pot utiliza ulterior pentru a declara una sau mai multe variabile. Cu alte cuvinte, programul definește mai întâi o structură și apoi declară variabile ce au ca tip acea structură. Pentru definirea unei structuri, programul apelează la cuvântul cheie *struct*, urmat de obicei de un nume și de o acoladă deschisă. După această acoladă deschisă se specifică tipul și numele pentru unul sau mai mulți *membri*. După precizarea ultimului membru urmează o acoladă închisă. Aici pot fi declarate eventuale variabile ce au ca tip structura definită, așa cum se vede aici:

```
struct nume {
  int nume_membru_1;      ├──── Declarațiile membrilor
  float nume_membru_2;
} variabila;  ├────────── Declarațiile de variabile
```

De exemplu, definiția următoare creează o structură numită *employee* care conține informații despre un angajat:

```
struct employee {
  char name[64];
  long employee_id;
  float salary;
  char phone[10];
  int office_number;
};
```

În acest exemplu, definiția structurii nu declară nici o variabilă având tipul structurii. După definirea unei structuri, programul poate declara variabile de tipul structurii prin intermediul numelui respectivei structuri (numit, uneori, *identificator de structură*), ca mai jos:

```
  ┌──────────────────────────────────────── Identificator
employee boss, worker, new_employee;
                   └──────────────────────── Declarații de variabile
```

În exemplul anterior, instrucțiunea creează trei variabile având ca tip structura *employee*. În examinarea diverselor programe C++, ați putea întâlni declarații în care identificatorul de structură este precedat de cuvântul cheie *struct*, ca aici:

```
struct employee boss, worker, new_employee;
```

Programarea în C impune utilizarea cuvântului cheie *struct*, așa că programatorii l-ar putea specifica din obișnuință. În C++, însă, precizarea acestui cuvânt cheie este opțională.

Utilizarea membrilor unei structuri

O structură vă permite gruparea informaţiilor, numite membri, în cadrul unei singure variabile. Atribuirea unei valori sau accesarea valorii unui membru se face cu ajutorul *operatorului punct* (.) din C++. De exemplu, instrucţiunea următoare atribuie valori mai multor membri ai unei variabile numită *worker* şi având tipul *employee*:

```
worker.employee_id = 12345;
worker.salary = 25000.00;
worker.office_number = 102;
```

Pentru accesarea unui membru de structură, specificaţi numele variabilei, urmat de operatorul punct şi de numele acelui membru (*nume_structura.membru*). Spre exemplu, instrucţiunile următoare atribuie variabilelor din program valorile aflate în diferiţi membri de structură:

```
identification = worker.employee_id;
salary = worker.salary;
office = worker.office_number;
```

Programul următor, *Employee.CPP*, ilustrează utilizarea unei structuri de tip *employee*:

```
#include <iostream.h>
#include <string.h>

void main(void)
  {
    struct employee {
      char name[64];
      long employee_id;
      float salary;
      char phone[10];
      int office_number;
    } worker;

    // Copiem un nume in sirul de caractere
    strcpy(worker.name, "John Doe");
    worker.employee_id = 12345;
    worker.salary = 25000.00;
    worker.office_number = 102;
```

```
    // Copiem un numar de telefon in sirul de caractere
    strcpy(worker.phone, "555-1212");

    cout << "Employee:   << worker.name << endl;
    cout << "Phone: " << worker.phone << endl;
    cout << "Employee id:   << worker.employee_id << endl;
    cout << "Salary:   << worker.salary << endl;
    cout << "Office: " << worker.office_number << endl;
}
```

După cum puteți vedea, programul poate atribui valori membrilor întreg și în virgulă mobilă ai structurii într-o manieră foarte simplă. Programul nu face altceva decât să utilizeze operatorul de atribuire pentru a atribui o valoare membrului corespunzător. Remarcați, totuși, folosirea funcției *strcpy* pentru copierea șirurilor de caractere în cazul membrilor *name* și *phone*. Dacă nu cumva inițializați explicit membrii unei structuri la declararea variabilei de tip structură, atunci va trebui să fixați valorile membrilor șir de caractere prin copierea șirurilor corespunzătoare. Veți învăța mai târziu în această lecție cum puteți inițializa membrii la declararea unei variabile structură.

Declararea variabilelor structură



Structurile din C++ permit programelor plasarea într-o singură variabilă a informațiilor înrudite ale căror tipuri pot diferi. O structură definește un șablon în scopul declarării ulterioare de variabile. Fiecare structură are un nume unic (numit uneori și *identificator*). Prin intermediul numelui unei structuri puteți să declarați variabile având ca tip acea structură. Programatorii numesc informațiile păstrate într-o structură, membri. Pentru utilizarea sau atribuirea unei valori de membru se folosește operatorul punct din C++, ca aici:

```
variabila.membru = o_valoare;
o_variabila = variabila.alt_membru;
```

Structuri și funcții

Dacă o funcție nu modifică o structură, atunci puteți transmite funcției respectiva structură prin numele său. De exemplu, programul următor, *Show_Emp.CPP*, folosește funcția *show_employee* pentru a afișa membrii unei structuri de tip *employee*:

```
#include <iostream.h>
#include <string.h>

struct employee {
   char name[64];
   long employee_id;
   float salary;
   char phone[10];
   int office_number;
   };

void show_employee(employee worker)
   {
     cout << "Employee:   << worker.name << endl;
     cout << "Phone:   << worker.phone << endl;
     cout << "Employee id: " << worker.employee_id << endl;
     cout << "Salary:   << worker.salary << endl;
     cout << "Office:   << worker.office_number << endl;
   }

void main(void)
   {
     employee worker;

     // Copiem un nume in sirul de caractere
     strcpy(worker.name, "John Doe");

     worker.employee_id = 12345;
     worker.salary = 25000.00;
     worker.office_number = 102;

     // Copiem un numar de telefon in sirul de caractere
     strcpy(worker.phone, "555-1212");
     show_employee(worker);
   }
```

Precum vedeți, programul transmite funcției *show_employee* variabila structură *worker* prin intermediul numelui acesteia. Apoi, în interiorul funcției, *show_employee* afișează membrii structurii. Remarcați, însă, că programul definește aici structura *employee* în afara programului principal și înaintea funcției *show_employee*. Deoarece funcția declară variabila *worker* ca având tipul *employee*, definiția structurii *employee* trebuie să preceadă această funcție.

Funcții care modifică membrii de structuri

După cum ați văzut, pentru ca o funcție să poată modifica un parametru, acel parametru trebuie transmis funcției prin adresă. Dacă o funcție trebuie să modifice valoarea unui membru de structură, acea structură trebuie transmisă funcției prin adresă. Transmiterea unei variabile structură prin adresă se face prin simpla precedare a numelui de variabilă cu operatorul de adresare (&) din C++, ca mai jos:

```
o_functie(&worker);
```

În cadrul unei funcții care modifică unul sau mai mulți membri veți fi nevoit să lucrați cu un pointer. Atunci când folosiți un pointer la o structură, calea cea mai simplă de a referi un membru al structurii este să apelați la sintaxa următoare:

```
variabila_pointer->membru = o_valoare;
```

Spre exemplu, programul următor, *Chg_Mbr.CPP*, transmite funcției *get_employee* o structură de tip *employee*, funcția solicită utilizatorului un număr de identificare al angajatului și apoi atribuie numărul citit membrului *employee_id* al structurii. Pentru modificarea membrului, funcția lucrează cu un pointer la structură:

```
#include <iostream.h>
#include <string.h>

struct employee {
   char name[64];
   long employee_id;
   float salary;
   char phone[10];
   int office_number;
   };

void get_employee_id(employee *worker)
   {
     cout << "Type in an employee id: ";
     cin >> worker->employee_id;
   }

void main(void)
   {
     employee worker;
```

```
    // Copiem un nume in sirul de caractere
    strcpy(worker.name, "John Doe");
    get_employee_id(&worker);

    cout << "Employee: " << worker.name << endl;
    cout << "Id: " << worker.employee_id << endl;
}
```

Așa cum se poate vedea, programul principal transmite funcției *get_employee* variabila structură *worker* prin adresă. În interiorul funcției, *get_employee_id* atribuie valoarea introdusă de utilizator membrului *employee_id*, folosind instrucțiunea următoare:

```
cin >> worker->employee_id;
```

Utilizarea pointerilor la structuri



Atunci când o funcție modifică un membru de structură, apelantul trebuie să transmită structura către funcție prin adresă. Funcția va utiliza, la rândul său, un pointer la structură. Accesarea în funcție a unui membru de structură se face sub următoarea formă:

```
valoare = variabila->membru;
variabila->alt_membru = o_valoare;
```

Inițializarea membrilor unei structuri

La declararea variabilelor structură, C++ vă permite inițializarea membrilor acelor variabile. De exemplu, declarația următoare creează și inițializează o variabilă structură de tip *carte*:

```
struct carte {
   char *titlu;
   float pret;
   char *autor;
} carte_calculator = { "Rescued by C++", 29.95, "Jamsa" };
```

Precum vedeți, instrucțiunea conține valorile inițiale ale variabilei între acolade, foarte asemănător cu maniera de inițializare a elementelor de vectori pe care ați utilizat-o în lecția 18.

Ce trebuie să știți

Structurile permit programelor să plaseze într-o aceeași variabilă informații înrudite, dar ale căror tipuri ar putea diferi. Printr-o astfel de combinare a datelor într-o variabilă unică, programele pot reprezenta mai bine obiectele alcătuite din două sau mai multe elemente, așa cum sunt angajații, cărțile etc. În lecția 21, „Despre uniuni", veți învăța să utilizați uniunile din C++ care, asemeni structurilor, conțin membri. Veți vedea, însă, că uniunile folosesc memoria foarte diferit în comparație cu structurile. O uniune poate reține o singură valoare la un moment dat, indiferent de numărul de membri. Dar înainte de a trece la lecția 21, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Structurile permit programelor să plaseze într-o singură variabilă elemente de informații înrudite, dar ale căror tipuri pot diferi.

☑ Programatorii numesc membri acele elemente de informație care compun o structură.

☑ O structură definește un șablon pe care programele îl pot utiliza pentru declararea de variabile.

☑ După definirea unei structuri, puteți folosi numele (identificatorul) acesteia pentru a declara variabile ce au ca tip respectiva structură.

☑ Pentru a atribui o valoare sau pentru a accesa valoarea unui membru de structură, programele apelează la operatorul punct din C++, sub forma *variabila.membru*.

☑ Dacă o funcție modifică valoarea unui membru de structură, programul trebuie să transmită funcției variabila structură prin adresă.

☑ Atunci când folosește un pointer la o structură, o funcție utilizează forma *variabila->membru* pentru a accesa un membru de structură.

Lecția 21

Despre uniuni

În lecția 20, „Păstrarea informațiilor înrudite în cadrul structurilor", ați învățat să grupați informații înrudite într-o singură variabilă prin intermediul structurilor C++. Odată cu creșterea complexității programelor, ar putea exista cazuri în care un program să presupună mai multe perspective diferite asupra unui același element de informație. În plus, s-ar putea ca un program să fie nevoit să lucreze cu două sau mai multe valori, utilizând, însă, o singură valoare dintre acestea la un moment dat. În astfel de situații, programele pot recurge la *uniuni* pentru a păstra date. În lecția de față veți învăța să creați și să utilizați uniuni pentru păstrarea de informații. După cum veți vedea, uniunile se aseamănă mult cu structurile despre care am discutat în lecția 20. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Uniunile din C++ sunt foarte similare structurilor, diferit fiind modul în care C++ le păstrează pe acestea în memorie – o uniune poate reține valoarea unui singur membru la un moment dat.
- O uniune este o structură de date care, asemeni unei structuri, este alcătuită din unul sau mai mulți membri.
- O uniune definește un șablon pe baza căruia programele pot apoi să declare variabile.
- Pentru accesarea unui membru de uniune, programele apelează la operatorul punct din C++.
- Pentru a modifica valoarea unui membru de uniune în cadrul unei funcții, programul trebuie să transmită acelei funcții variabila uniune prin adresă.
- O *uniune anonimă* este o uniune care nu are un nume (un identificator).

Așa cum veți vedea, uniunile sunt asemănătoare structurilor din C++, însă modul în care C++ stochează uniunile diferă de modul de stocare al structurilor.

Cum sunt stocate uniunile în C++

Într-un program, o uniune C++ este foarte similară cu o structură. Spre exemplu, instrucțiunea următoare definește o uniune numită *distance* care conține doi membri:

```
union distance {
   int miles;
   long meters;
};
```

Ca și în cazul structurilor, definiția uniunii nu alocă memorie. În schimb, definiția oferă un șablon pe baza căruia programele pot declara mai apoi o variabilă uniune. Declararea unei variabile uniune se poate face sub oricare din cele două forme de mai jos:

```
union distance {
   int miles;
   long meters;
} japan, germany, france;
```

```
union distance{
   int miles;
   long meters;
   };

distance japan, germany, france;
```

Precum vedeți, uniunea *distance* conține doi membri, *miles* și *meters*. Declarațiile creează variabilele care vă permit păstrarea distanțelor până la țările specificate. Asemeni unei structuri, programul poate atribui o valoare oricăruia dintre membri. Spre deosebire de o structură, însă, programul poate atribui la un moment dat o valoare unui singur membru. Cu alte cuvinte, atunci când programul atribuie o valoare unui membru al uniunii, noua valoare suprascrie eventuala valoare pe care programul a atribuit-o anterior aceluiași sau chiar unui alt membru al uniunii. La declararea unei uniuni, compilatorul de C++ alocă memorie pentru a putea reține cel mai mare membru al uniunii. În cazul uniunii *distance*, compilatorul alocă memoria necesară pentru a păstra o valoare de tip *long*, așa cum o arată figura 21.1.

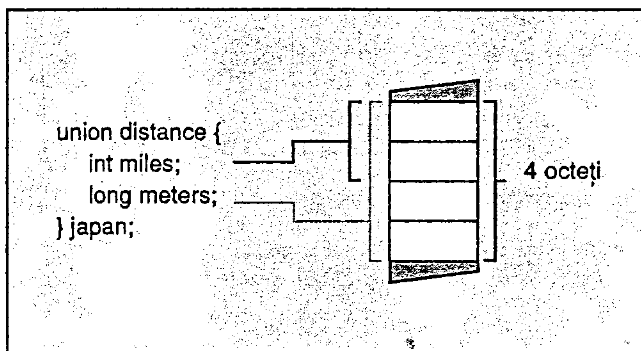


Figura 21.1 *C++ alocă memoria necesară pentru a reține cel mai mare membru al uniunii.*

Să presupunem că programul atribuie o valoare membrului *miles*, ca mai jos:

```
japan.miles = 12123;
```

Dacă programul va atribui ulterior o valoare membrului *meters*, noua valoare va suprascrie valoarea atribuită membrului *miles*.

Programul următor, *UseUnion.CPP*, ilustrează utilizarea uniunii *distance*. Programul atribuie mai întâi o valoare membrului *miles* și apoi afișează acea valoare. În continuare, pro-

gramul atribuie o valoare membrului *meters*. În acest moment, prin atribuirea unei valori membrului *meters*, programul suprascrie valoarea membrului *miles*, ca în continuare:

```
#include <iostream.h>

void main(void)
  {
    union distance {
      int miles;
      long meters;
    } walk;

    walk.miles = 5;

    cout << "A distance walked in miles is "
      << walk.miles << endl;

    walk.meters = 10000;

    cout << "A distance walked in meters is "
      << walk.meters << endl;
  }
```

După cum puteți observa, programul accesează membrii uniunii cu ajutorul unei notației cu punct similară celei pe care ați utilizat-o pentru accesarea membrilor de structură în lecția 20.

Uniunile rețin la un moment dat valoarea unui singur membru



O uniune este o structură de date care, asemeni unei structuri, permite programelor să păstreze elemente de informație înrudite într-o singură variabilă. Spre deosebire, însă, de o structură, o uniune reține la un moment dat valoarea unui singur membru. Cu alte cuvinte, atunci când atribuiți o valoare unui membru de uniune, aceasta suprascrie orice atribuire anterioară. O uniune definește un șablon pe baza căruia programele pot să declare apoi variabile. Atunci când întâlnește definiția unei uniuni, compilatorul de C++ alocă exact memoria necesară pentru a păstra cel mai mare membru al uniunii.

Despre uniunile anonime din C++

O *uniune anonimă* este o uniune care nu are un nume. C++ oferă aceste uniuni anonime în scopul simplificării utilizării membrilor de uniune în programele care utilizează uniuni pentru a economisi memorie sau pentru a crea un alias al unei valori. De exemplu, să

presupunem că un program necesită două variabile, *miles* și *meters*. De asemenea, să presupunem că programul folosește la orice moment dat doar o singură variabilă dintre acestea. Utilizând uniunea *distance* despre care am discutat, programul ar apela apoi la membrii acestei uniuni, *nume.miles* și *nume.meters*. Ca alternativă, instrucțiunea următoare creează o uniune anonimă (fără nume):

```
union {
   int miles;
   long meters;
};
```

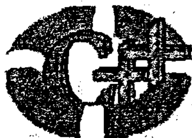
După cum vedeți, declarația nu precizează nici un nume de uniune și nu declară nici o variabilă de tipul acestei uniuni. Programul, însă, va putea să refere numele membrilor *miles* și *meters* fără a mai apela la notația cu punct. Programul următor, *Anonym.CPP*, creează o uniune anonimă care conține membrii *miles* și *meters*. Așa cum se vede, programul tratează acești membri ca și cum ar fi variabile obișnuite. Diferența dintre membri și o variabilă obișnuită este, însă, că atribuirea unei valori oricăruia dintre membri duce la suprascrierea valorii celuilalt membru:

```
#include <iostream.h>

void main(void)
  {
    union {
       int miles;
       long meters;
    };
    miles = 10000;
    cout << "The value of miles is " << miles << endl;
    meters = 150000L;
    cout << "The value of meters is " << meters << endl;
  }
```

După cum puteți remarca, prin utilizarea unei uniuni anonime, programul poate economisi memorie fără a mai purta povara specificării numelui de uniune și a notației cu punct pentru accesarea valorilor de membri.

Uniunile anonime permit programelor să economisească memorie



O uniune anonimă este o uniune fără nume. Uniunile anonime oferă programelor posibilitatea de a economisi memorie fără a mai recurge la notația cu punct. Instrucțiunile următoare definesc o uniune anonimă care poate păstra două șiruri de caractere:

```
union {
  char nume_scurt[13];
  char nume_lung[255];
};
```

Ce trebuie să știți

În lecția de față ați învățat să creați și să utilizați uniuni în cadrul programelor. După cum ați aflat, forma unei uniuni este foarte similară cu cea a unei structuri. Modul în care C++ stochează uniunile diferă, însă, de modul de stocare a unei structuri. În lecția 11, „Modificarea valorilor parametrilor", ați văzut că pentru ca o funcție să poată modifica un parametru este necesar ca programul să transmită acel parametru către funcție cu ajutorul unui pointer (sau adresă de memorie). Începând cu lecția 11, programele au utilizat pointeri pentru vectori și pentru șiruri de caractere. Lecția 22, „Despre pointeri", vă va oferi o altă perspectivă asupra operațiilor cu pointeri în C++. Dar înainte de a trece la lecția 22, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ La declararea unei uniuni, compilatorul de C++ alocă exact memoria necesară pentru a reține cel mai mare membru al uniunii.
- ☑ Definiția unei uniuni nu duce la alocarea de memorie; ea oferă, în schimb, un șablon pe baza căruia programele pot declara mai târziu variabile.
- ☑ Programele accesează membrii de uniune prin intermediul notației cu punct. Atunci când un program atribuie o valoare unui membru de uniune, orice valoare atribuită anterior unui alt membru al uniunii este suprascrisă. Cu alte cuvinte, o uniune poate să rețină la un moment dat valoarea unui singur membru.
- ☑ O uniune anonimă este o uniune care nu are un nume. Atunci când un program declară o uniune anonimă, membrii acestei uniuni pot fi utilizați ca orice variabilă obișnuită, fără necesitatea de a mai recurge la notația cu punct.

Lecția 22

Despre pointeri

Așa cum ați aflat, programele C++ păstrează variabilele în memorie. Un pointer este o adresă de memorie care „indică" sau referă o locație anume. În lecția 11, „Modificarea valorilor parametrilor", ați învățat că pentru modificarea unui parametru în cadrul unei funcții este necesar ca programul să transmită funcției adresa respectivului parametru (un pointer). Funcția, la rândul ei, utilizează apoi o variabilă pointer pentru a accesa locația de memorie corespunzătoare. Mai multe din programele pe care le-ați creat pe parcursul ultimelor câteva lecții au folosit pointeri la parametri. De asemenea, atunci când programele lucrează cu șiruri de caractere și vectori, este uzual ca aceste programe să folosească pointeri pentru manipularea elementelor din vectori. Deoarece pointerii sunt utilizați atât de frecvent, este foarte important să înțelegeți modul de folosire al acestora. Din acest motiv, lecția de față se oprește din nou asupra pointerilor. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru simplitate (pentru reducerea codului), multe programe tratează un șir de caractere ca un pointer și manipulează conținutul șirului cu ajutorul operațiilor cu pointeri.
- La incrementarea unei variabile pointer (o variabilă ce reține o adresă), C++ incrementează automat adresa cu o valoare adecvată (1 octet pentru *char*, 2 octeți pentru *int*, patru octeți pentru *float*, etc.), astfel încât pointerul să indice următoarea valoare pe care o are tipul pointerului.
- Programele pot utiliza pointeri pentru a lucra cu vectori de valori întregi sau în virgulă mobilă.

Operațiile cu pointeri sunt foarte uzuale în C++. Acordați-vă timpul necesar pentru a experimenta cu programele prezentate în această lecție.

Utilizarea unui pointer la un șir de caractere

După cum ați aflat, un pointer conține o adresă de memorie. Atunci când programele transmit vectori (un șir de caractere, de pildă) către funcții, C++ transmite ca parametru adresa primului element al vectorului. Prin urmare, este foarte uzual ca funcțiile să folosească pointeri la șiruri de caractere. Pentru a declara un pointer la un șir de caractere, este suficient ca funcția să preceadă numele variabilei cu un asterisc, așa cum se vede aici:

```
void o_functie(char *sir);
```

Asteriscul care precede numele variabilei informează C++ că respectiva variabilă va păstra o adresă de memorie, un pointer. Programul următor, *Ptr_str.CPP*, folosește în cadrul

funcției *show_string* un pointer la un șir de caractere pentru a afișa conținutul șirului caracter cu caracter:

```
#include <iostream.h>

void show_string(char *string)
  {
    while (*string != '\0')
      {
        cout << *string;
        string++;
      }
  }
void main(void)
  {
    show_string("Rescued By C++!");
  }
```

Priviți cu atenție bucla *while* din funcția *show_string*. Condiția din buclă (**string != '\0'*) testează dacă litera curentă indicată de pointerul *string* este diferită de NULL, care indică ultimul caracter al șirului. În cazul în care caracterul este diferit de NULL, bucla afișează acest caracter cu ajutorul *cout*. Apoi, instrucțiunea *string++* incrementează pointerul *string* astfel încât acesta va indica următorul caracter din șir. Atunci când pointerul *string* ajunge să indice caracterul NULL, funcția a afișat deja șirul și bucla se încheie.

Să presupunem, de pildă, că șirul transmis funcției se află în memoria calculatorului la adresa 1000. De fiecare dată când funcția incrementează pointerul *string*, acesta indică următorul caracter (adresa 1001, 1002, 1003 etc.), așa cum o ilustrează figura 22.

Un alt exemplu

După cum tocmai ați văzut, funcțiile pot parcurge caracterele unui șir prin intermediul unui pointer până la întâlnirea caracterului NULL. Programul următor, *Ptr_Len.CPP*, utilizează în funcția *string_length* un pointer la un șir de caractere pentru a determina numărul de caractere din șir:

```
#include <iostream.h>

int string_length(char *string)
  {
    int length = 0;
```

```
    while (*string != '\0')
    {
      length++;
      string++;
    }
    return(length);
  }

void main(void)
  {
    char title[] = "Rescued By C++"
    cout << title << " contains " << string_length(title)
      << " characters"
  }
```

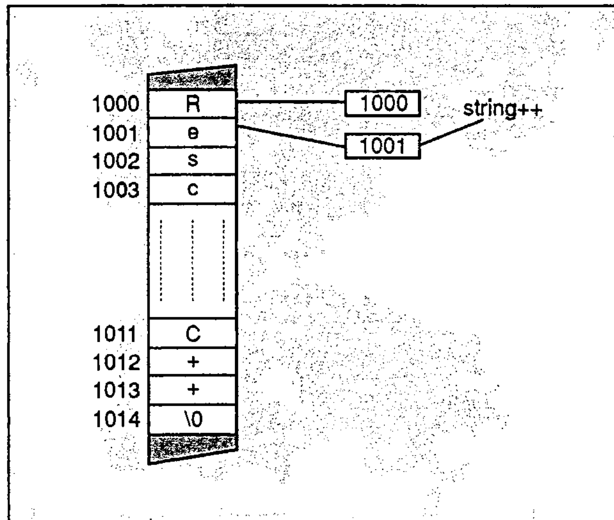


Figura 22 *Parcurgerea unui șir cu ajutorul unui pointer.*

Precum vedeți, funcția *string_length* parcurge iterativ caracterele șirului până la întâlnirea caracterului NULL.

Incrementarea unui pointer la un şir de caractere



Atunci când un program transmite un vector unei funcţii, C++ transmite adresa de memorie a primului element al vectorului. Cu ajutorul unei variabile pointer, funcţia poate parcurge conţinutul vectorului prin incrementarea valorii pointerului. De exemplu, să presupunem că un program transmite unei funcţii şirul de caractere „Hello". În cadrul funcţiei, variabila pointer indică iniţial către locaţia de memorie care conţine litera 'H'. Atunci când funcţia incrementează pointerul, acesta va indica la locaţia de memorie ce conţine litera 'e'. Pe măsură ce funcţia continuă incrementarea valorii pointerului, acesta va indica pe rând fiecare literă din şir şi, în cele din urmă, caracterul NULL.

Înlăturarea instrucţiunilor inutile

Pentru a identifica sfârşitului unui şir de caractere, fiecare dintre programele anterioare foloseau următoarea buclă *while*:

```
while (*string != '\0')
```

După cum am discutat, caracterul NULL ('\0') are codul ASCII 0. Deoarece C++ reprezintă prin 0 valoarea de fals, programele pot rescrie bucla de mai sus sub forma:

```
while (*string)
```

În acest exemplu, condiţia este evaluată ca adevărată şi bucla continuă atât timp cât caracterul indicat de pointer este diferit de 0 (NULL).

În lecţia 6, „Efectuarea de operaţii elementare", aţi văzut că operatorul de incrementare postfixat din C++ vă permite să utilizaţi valoarea unei variabile şi, apoi, să incrementaţi acea valoare. Multe programe C++ utilizează operatorii de incrementare şi decrementare postfixaţi pentru a parcurge vectorii cu ajutorul pointerilor. De exemplu, prin utilizarea operatorului de incrementare postfixat, cele două bucle *while* de mai jos sunt identice:

```
while (*string)
  {
    cout << *string;
    string++;
  }
```

```
while (*string)
  cout << *string++;
```

Instrucţiunea *cout << *string++;* determină C++ să afişeze caracterul indicat de către *string* şi apoi să incrementeze valoarea curentă a pointerului *string* astfel încât acesta să indice caracterul următor. Prin intermediul acestor tehnici, programul *SmartPtr.CPP* ilustrează noi implementări ale funcţiilor *show_string* şi *string_length*:

```
#include <iostream.h>

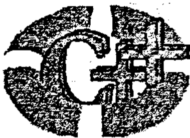
void show_string(char *string)
   {
     while (*string)
        cout << *string++;
   }

int string_length(char *string)
   {
     int length = 0;
     while (*string++)
        length++;
     return(length);
   }

void main(void)
   {
     char title[] = "Rescued By C++";
     show_string(title)
     cout << " contains  <<
        string_length(title) << " characters";
   }
```

Pe măsură ce veți întâlni diferite funcții C++ care manipulează șirurile de caractere prin intermediul pointerilor, aceste funcții vor folosi probabil astfel de notații prescurtate.

Parcurgerea unui șir de caractere



Una dintre cele mai uzuale întrebuințări ale pointerilor în cadrul programelor C++ este parcurgerea șirurilor de caractere. În scopul reducerii efortului de programare, multe programe vor recurge la următoarele instrucțiuni pentru a efectua parcurgerea unui șir:

```
while (*sir)
   {
        // instructiuni
        sir++; // indica urmatorul caracter
   }
```

Funcția următoare, *string_uppercase*, utilizează pointeri pentru a transforma caracterele șirului în majuscule:

```
char *string_uppercase(char string)
  {
    char *starting_address = string; // adresa
                                     // pentru string[0];
    while (*string)
      {
        if ((*string >= 'a') && (*string <= 'z'))
          *string = *string - 'a' + 'A';
        string++;
      }
    return(starting_address);
  }
```

Funcția *string_uppercase* reține și apoi întoarce adresa de început a șirului, ceea ce permite programelor să folosească această funcție în felul următor:

```
cout << string_uppercase("Hello, world!") << endl;
```

Prin întoarcerea adresei de început a șirului către apelant, codul poate utiliza adresa furnizată de funcție pentru a accesa șirul. În exemplul de mai sus, codul folosește adresa șirului în cadrul fluxului *cout* pentru a afișa conținutul respectivului șir.

Utilizarea pointerilor pentru alte tipuri de vectori

Deși pointerii sunt cel mai uzual utilizați în cazul șirurilor de caractere, aceștia pot fi folosiți și pentru alte tipuri de vectori. Spre exemplu, programul care urmează, *PtrFloat.CPP*, folosește un pointer la un vector de tip *float* pentru a afișa valorile în virgulă mobilă conținute:

```
#include <iostream.h>

void show_float(float *array,
  int number_of_elements)
  {
    int i;
    for (i = 0; i < number_of_elements; i++)
```

```
        cout << *array++ << endl;
    }

void main(void)
    {
      float values[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

      show_float(values, 5);
    }
```

După cum puteți vedea, bucla *for* din funcția *show_float* utilizează valoarea indicată de pointerul *array* și apoi incrementează valoarea pointerului pentru a indica elementul următor. În acest exemplu, programul trebuie să transmită un parametru care indică numărul de elemente din vector deoarece, spre deosebire de șirurile de caractere, vectorii de tip *float* (sau *int*, *long*, etc.) nu folosesc un caracter NULL pentru a marca ultimul element.

Despre aritmetica cu pointeri

Așa cum ați aflat, programele pot folosi pointeri la orice tip de vector. În programul anterior, de pildă, funcția *show_float* incrementa un pointer pentru a parcurge un vector de tip *float*. Precum știți, un pointer indică o locație de memorie care conține o valoare de un anume tip, cum ar fi *char*, *int* sau *float*. La parcurgerea unui vector prin intermediul unui pointer, funcția incrementează pointerul pentru a trece de la o valoare la următoarea. Pentru ca un pointer să poată indica următoarea valoare din vector, C++ trebuie să urmărească dimensiunea fiecărei valori (în octeți), astfel încât să știe cu cât trebuie incrementată variabila pointer. De exemplu, pentru a deplasa un pointer la următorul caracter dintr-un șir, C++ trebuie să incrementeze valoarea pointerului cu un octet. Pentru a trece, însă, la următoarea valoare dintr-un vector de tip *int*, C++ trebuie să incrementeze pointerul cu doi octeți (valorile de tip *int* necesită doi octeți de memorie). În cazul valorilor de tip *float*, C++ incrementează pointerul cu patru octeți. Cunoscând tipul valorii indicate de către pointer, C++ știe cu cât trebuie incrementată valoarea pointerului. În program este suficient să folosiți un operator de incrementare, precum *pointer++*, pentru a efectua incrementarea unei variabile pointer. C++ va incrementa, de fapt, valoarea conținută de pointer (adresa de memorie) cu cantitatea corespunzătoare.

Ce trebuie să știți

Programele C++ folosesc frecvent pointeri, mai cu seamă pentru manipularea șirurilor de caractere. Deoarece programatorii utilizează adesea operații cu pointeri, lecția de față v-a oferit o nouă perspectivă asupra operațiilor cu pointeri din C++. În lecția 23, „O introducere în clasele C++", veți începe să utilizați facilitățile de programare orientată spre obiect din C++. Pentru început veți crea clase care sunt asemănătoare structurilor. Programele vor defini un obiect cu ajutorul unei clase, ca de pildă *fisier*. În cadrul clasei veți specifica funcțiile care manipulează obiectul respectiv, precum *tipareste_fisier* sau

sterge_fisier. Dar înainte de a trece la lecţia 23, asiguraţi-vă că aţi reţinut următoarele aspecte importante:

☑ Pointerii conţin o adresă de memorie. Atunci când transmiteţi un vector unei funcţii, C++ transmite de fapt adresa primului element al vectorului.

☑ Prin incrementarea valorii unui pointer puteţi determina acel pointer să indice următorul element dintr-un vector.

☑ Funcţiile care manipulează şiruri de caractere cu ajutorul pointerilor parcurg, de regulă, şirul până când caracterul indicat de pointer este NULL.

☑ Atunci când utilizaţi pointeri pentru alte tipuri de vectori, funcţiile trebuie să cunoască numărul de elemente din vector sau o valoare specială de sfârşit.

☑ Atunci când utilizaţi pointeri pentru alte tipuri de vectori, C++ incrementează automat (în culise) pointerul (adresa de memorie) cu o cantitate corespunzătoare, astfel încât acel pointer să indice următorul element din vector.

C++

Abstract:

A basic introduction to Object Oriented Programming with C++. An ability to design and write programs in a modern procedural language like Ada, Modula-2 or Pascal is assumed. C experience is not required. The coverage of the language is comprehensive. Classes, single and multiple inheritance, virtual functions and templates are discussed. However, exception handling is not described. Compiler specific features are not used. This material is suitable for use with any quality C++ compiler.

CONCENTRATED C++
An Introduction to the Language (2nd Edition)

Adrian P. Robson
The University of Northumbria at Newcastle



http://cmuwww.unn.ac.uk/~adrian/ c++/cppaper/cppaper.html

CĂRȚI BORLAND C++

BorlandOnline

Borland C++

IS SUPPORTED BY THESE TITLES FROM LEADING BOOK PUBLISHERS

http://www.borland.com/borlandcpp/ books/index.html

BIBLIOTECA VIRTUALĂ C++

The C++ Virtual Library

Recent Changes -- Mail additions to this list to
You can also search for a keyword on this server.

The HTML formatted version on the draft C++ standard.

Getting Started
Documents and sources on C++ and OOP. The draft C++ standard is now generally available. Please read the informal announcement.
Emacs
Customizable environment for Emacs editors
Learning C++
Virtual courses and tutorials.
Newsgroups

http://www.desy.de/ftp/pub/ userwww/projects/C++.html

PROGRAMARE ÎN C

Next: Copyright

Programming in C

© David Marshall 1994

Search for Keywords in C Notes

http://www.cm.cf.ac.uk/Dave/C/ CE.html

RESURSE VISUAL C++

AUL **Visual C++**

News | Resources on the Web | FAQ's

Microsoft's For Developers Only
MS's Visual C++ Page
Internet Resources for Windows Developers
Win32 Development Knowledge Base
Visual C++ Developers Journal

Newsgroups

http://www.aul.fiu.edu/tech/ visualc.html

REGULI ȘI RECOMANDĂRI

Programming in C++, Rules and Recommendations

FN/Mats Henricson and Erik Nyquist

Copyright (C) 1990-1992 by

Ellemtel Telecommunication Systems Laboratories
Box 1505
125 25 Älvsjö
Sweden
Tel: int + 46 8 727 30 00

http://www.rhi.hi.is/~harri/ cpprules.html

LIMBAJUL DE PROGRAMARE C++

The C++ Programming Language



http://www.research.att.com/~bs/ C++.html

INFORMAŢII DESPRE IOSTREAM

Information About IOstreams

http://www.informatik.uni-konstanz.de/ %7Ekuehl/c++/iostream

ÎNTREBĂRI DESPRE C++

comp.std.c++ frequently asked questions

http://reality.sgi.com/austern/std-c++/ faq.html

COPII STANDARD C++

The C++ Draft Standard

http://www-leland.stanford.edu/ ~iburell/cpp/std.html

LEGĂTURI C++

C++ And Object Oriented Resource Page

http://www.canisius.edu/~duchan/ duchan2.html

GHIDUL MUMIT'S STL NEWBIE

Mumit's STL Newbie guide

http://www.xraylith.wisc.edu/~khan/ software/stl/STL.newbie.html

PARTEA a IV-a

Utilizarea claselor în C++

Programarea orientată spre obiect se concentrează asupra obiectelor (lucrurilor) care compun un sistem. Ați putea avea de a face, de exemplu, cu un obiect fișier, un obiect angajat și așa mai departe. Fiecare obiect reține informații asociate, precum un nume de fișier, numele sau numărul de identificare al unui angajat, eventual salariul acestuia. Din această perspectivă, obiectele sunt asemănătoare structurilor din C++. Obiectele definesc, însă, și o serie de operații pe care programele le pot efectua asupra datelor din obiecte. În cazul unui obiect fișier, de pildă, ați putea să tipăriți, să ștergeți sau chiar să copiați fișierul. Analog, pentru un obiect angajat ați putea să tipăriți, să avansați sau chiar să concediați un obiect anume. C++ utilizează clase pentru a reține datele unui obiect și funcțiile care pot fi aplicate acelor date. Această parte a cărții studiază în amănunt clasele din C++. Lecțiile cuprinse aici sunt următoarele:

Lecția 23

O introducere în clasele C++

Clasele reprezintă principalul instrument pentru programarea orientată spre obiect din C++. Așa cum veți învăța în această lecție, o clasă este foarte asemănătoare unei structuri prin faptul că grupează membri ce corespund datelor unui obiect, precum și funcții (numite *metode*) care operează asupra datelor. După cum veți vedea, un obiect este un *lucru*, precum un telefon, un fișier sau o carte. O clasă C++ permite programelor să definească atributele obiectului (caracteristicile sale). În cazul unui obiect *telefon*, clasa ar putea conține date membru, precum numărul de telefon și tipul acestuia (în modul „pulse" sau „tone"), și funcții care acționează asupra telefonului, cum ar fi *formeaza*, *raspunde* și *inchide*. Gruparea într-o singură variabilă a datelor și codului unui obiect duce la simplificarea programării și la creșterea reutilizabilității codului (capacitatea de a folosi același cod într-un alt program). Lecția de față reprezintă o introducere în clasele C++. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru definirea unei clase, programul trebuie să precizeze numele clasei, membrii de date ai clasei și funcțiile clasei (metode).
- Definiția unei clase oferă un șablon pe baza căruia programele pot crea obiecte de tipul acelei clase, destul de similar cu modul în care programele creează variabile de tip *int*, *char*, etc.
- Atribuirea în programe a valorilor pentru membrii de date ai claselor se face prin intermediul operatorului punct.
- Invocarea de către program a funcțiilor membru ale unei clase (*metode*) se face prin intermediul operatorului punct.

Despre obiecte și programarea orientată spre obiect

Din cel mai simplu punct de vedere, un *obiect* este un lucru. Atunci când creați un program, acesta folosește de regulă variabile pentru a păstra informații despre lucruri reale, cum ar fi angajați, cărți sau chiar fișiere. În programarea orientată spre obiect, în centrul atenției se află lucrurile care compun un sistem și operațiile pe care trebuie să le efectuați asupra acelor lucruri. Luând ca exemplu un obiect fișier, ați putea avea operații care tipăresc, afișează, șterg sau modifică fișierul. Definirea în C++ a obiectelor se face cu ajutorul claselor. Atunci când definiți o clasă, scopul dumneavoastră este să includeți în aceasta cât mai multe informații posibile despre obiectul corespunzător. În acest fel devine posibil să „preluați" o clasă pe care ați creat-o pentru un program și să o folosiți în multe alte programe.

O clasă C++ permite programelor să grupeze date și funcții ce efectuează operații asupra datelor. Majoritatea cărților și articolelor despre programarea orientată spre obiect se referă

la funcțiile unei clase ca la *metode*. Asemenea unei structuri, o clasă C++ trebuie să aibă un nume unic, urmat de o acoladă deschisă, unul sau mai mulți membri și o acoladă închisă, așa cum se vede aici:

```
class nume_clasa {
  int data_membru;                    // Data membru
  void afiseaza_membru(int);          // Functie membru
};
```

După ce definiți o clasă, puteți să declarați variabile de tipul acelei clase (numite *obiecte*) ca mai jos:

```
nume_clasa obiect_unu, obiect_doi, obiect_trei;
```

Definiția următoare creează o clasă *employee* care conține definițiile variabilelor de date și ale metodelor:

```
class employee {
  public:
  char name[64];
  long employee_id;
  float salary;
  void show_employee(void)
    {
      cout << "Name: " << name << endl;
      cout << "Id: " << employee_id << endl;
      cout << "Salary: " << salary << endl;
    };
};
```

În acest exemplu, clasa *employee* conține trei variabile membru și o funcție membru. Remarcați utilizarea etichetei *public* în cadrul definiției clasei. Așa cum veți afla din lecția 24, „Despre datele publice și private", membrii claselor pot fi *privați* sau *publici*, ceea ce determină modul în care programele pot accesa respectivii membri. Toți membrii din exemplul de față sunt *publici*, ceea ce înseamnă că programul poate accesa orice membru prin intermediul operatorului punct. După definirea unei clase în program puteți declara obiecte (variabile) de tipul acelei clase, așa cum se vede mai jos:

```
employee worker, boss, secretary;
```

Numele clasei

Variabile clasă (obiecte)

Programul următor, *EmpClass.CPP*, creează două obiecte *employee*. Cu ajutorul operatorului punct, programul atribuie valori datelor membru. Este apelat apoi membrul *show_employee* pentru a afişa informaţii despre un angajat, ca mai jos:

```
#include <iostream.h>
#include <string.h>

class employee {
  public:
  char name[64];
  long employee_id;
  float salary;
  void show_employee(void)
    {
      cout << "Name: " << name << endl;
      cout << "Id: " << employee_id << endl;
      cout << "Salary: " << salary << endl;
    };
};

void main(void)
  {
    employee worker, boss;

    strcpy(worker.name, "John Doe");
    worker.employee_id = 12345;
    worker.salary = 25000;

    strcpy(boss.name, "Happy Jamsa");
    boss.employee_id = 101;
    boss.salary = 101101.00;

    worker.show_employee();
    boss.show_employee();
  }
```

După cum puteţi vedea, programul declară două obiecte *employee*, *worker* şi *boss*, după care utilizează operatorul punct pentru a atribui valori membrilor şi invoca funcţia *show_employee*.

Despre obiecte



Cele mai multe programe C++ lucrează cu reprezentări ale unor entități din realitate, numite și obiecte. Dintr-o perspectivă simplistă, un obiect este un lucru, precum o pisică, un câine, un ceas și așa mai departe. Un obiect are, în mod normal, o serie de atribute și o serie de operații pe care programul le poate aplica asupra atributelor. Spre exemplu, în cazul unui obiect ceas, printre atribute s-ar putea afla ora curentă și ora pentru alarmă. Operațiile pe care programul le-ar putea efectua asupra ceasului includ fixarea orei, fixarea alarmei sau dezactivarea alarmei. Atunci când programele au la bază programarea orientată spre obiect, în centrul atenției se află obiectele și operațiile efectuate asupra acestor obiecte.

Declararea metodelor unei clase în exteriorul clasei

În clasa *employee* de mai devreme, funcțiile erau definite chiar în cadrul clasei (numindu-se *funcții inline*). Cum funcțiile pot deveni întinse, definirea lor în interiorul clasei aglomerează definiția de clasă. Ca alternativă, puteți să plasați în cadrul clasei prototipul unei funcții și apoi să definiți funcția undeva, în exteriorul acelei clase. În acest caz, definiția clasei conținând prototipul de funcție devine următoarea:

```
class employee {
  public:
  char name[64];
  long employee_id;
  float salary;
  void show_employee(void); ├──────── Prototipul funcției
};
```

Deoarece clase diferite ar putea folosi funcții cu același nume, este necesară precedarea numelor de funcții declarate în exteriorul claselor cu numele clasei, urmat de *operatorul de rezoluție globală* (::). În cazul nostru, definiția funcției *show_employee* devine următoarea:

```
void employee::show_employee(void)        ── Numele clasei
  {                                       ── Numele membrului
    cout << "Name: " << name << endl;
    cout << "Id: " << employee_id << endl;
    cout << "Salary: " << salary << endl;
  };
```

După cum puteţi observa, codul precedă definiţia funcţiei cu numele clasei (*employee*) şi operatorul de rezoluţie globală (::). Programul următor, *ClassFun.CPP*, deplasează definiţia funcţiei *show_employee* în exteriorul clasei, utilizând operatorul de rezoluţie globală pentru a specifica numele clasei:

```
#include <iostream.h>
#include <string.h>

class employee {
  public:
    char name[64];
    long employee_id;
    float salary;
    void show_employee(void);
};

void employee::show_employee(void)
  {
    cout << "Name: " << name << endl;
    cout << "Id: " << employee_id << endl;
    cout << "Salary: " << salary << endl;
  };

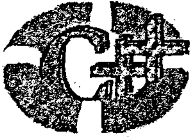
void main(void)
  {
    employee worker, boss;

    strcpy(worker.name, "John Doe");
    worker.employee_id = 12345;
    worker.salary = 25000;

    strcpy(boss.name, "Happy Jamsa");
    boss.employee_id = 101;
    boss.salary = 101101.00;

    worker.show_employee();
    boss.show_employee();
  }
```

Despre metodele claselor



Clasele C++ permit programelor să grupeze într-o singură variabilă datele unui obiect și funcțiile (metodele) care operează asupra acestor date. Pentru definirea metodelor unui obiect aveți două posibilități. Puteți, pe de o parte, să includeți întreg codul funcției în interiorul definiției de clasă. Deși această includere în definiția clasei a codului asociat obiectului pare convenabilă, odată ce clasele vor spori în complexitate și vor avea mai multe metode, instrucțiunile funcțiilor vor aglomera definiția de clasă. Acesta este motivul pentru care multe programe plasează instrucțiunile funcțiilor în exteriorul clasei. Programele includ în definiția clasei prototipurile de funcții care indică numele funcției, tipul întors și tipurile parametrilor.

Pentru a defini o funcție în exteriorul unei definiții de clasă, programul trebuie să preceadă definiția funcției cu numele clasei urmat de operatorul de rezoluție globală, așa cum se vede aici:

```
tip_intors nume_clasa::nume functie(parametri)
  {
    // Instructiuni
  }
```

Un alt exemplu

Programul care urmează, *Pedigree.CPP*, creează o clasă *dog* care conține mai multe câmpuri de date și o funcție *show_breed*. Funcția clasei este definită aici în exteriorul definiției de clasă. Programul creează apoi trei obiecte *dog* și afișează informații despre fiecare câine, așa cum se vede în continuare:

```
#include <iostream.h>
#include <string.h>

class dogs {
  public:
    char breed[64];
    int average_weight;
    int average_height;
    void show_breed(void);
};
```

```
void dogs::show_breed(void)
  {
    cout << "Breed: " << breed << endl;
    cout << "Average Weight: " << average_weight << endl;
    cout << "Average Height: " << average_height << endl;
  }

void main(void)
  {
    dogs happy, matt;

    strcpy(happy.breed, "Dalmatian");
    happy.average_weight = 58;
    happy.average_height = 24;

    strcpy(matt.breed, "Shetland Sheepdog");
    matt.average_weight = 22;
    matt.average_height = 15;

    happy.show_breed();
    matt.show_breed();
  }
```

Ce trebuie să știți

Programele C++ folosesc clase în mod frecvent. Pe scurt, o clasă permite programelor să grupeze într-o singură variabilă datele unui obiect și metodele (funcțiile) care operează asupra acelor date. Precum vedeți, clasele sunt destul de asemănătoare cu structurile despre care am discutat în lecția 20, „Păstrarea informațiilor înrudite în cadrul structurilor". Clasele C++ reprezintă baza programării orientate spre obiect. Lecțiile care urmează vor detalia diversele facilități oferite de clase. Așa cum am menționat în această lecție, eticheta *public* ce apare în definițiile de clasă face ca membrii clasei să fie accesibili oriunde în program. În lecția 24 veți afla mai multe informații privind membrii *privați* și *publici* ai unei clase. Dar înainte de a trece la lecția 24, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Sub cea mai simplă formă, un obiect este un lucru asupra căruia programul efectuează diferite operații.
- ☑ Programele C++ reprezintă obiectele prin intermediul claselor.

- ☑ O clasă, asemeni unei structuri, conține membri. Membrii unei clase pot reține informații (date) sau pot fi funcții (metode) care operează asupra datelor.
- ☑ Fiecare clasă are un nume unic.
- ☑ După definirea unei clase puteți să declarați obiecte ale acelei clase, folosind ca tip numele clasei.
- ☑ Programele accesează membrii unei clase (fie date, fie funcții) prin intermediul operatorului punct.
- ☑ Programele pot defini o funcție din clasă în interiorul sau în exteriorul definiției de clasă. În cazul în care definiți o funcție în exteriorul definiției de clasă, trebuie să specificați numele clasei respective și să utilizați operatorul de rezoluție globală, ca de pildă *clasa::functie.*

Lecția 24

Despre datele publice și private

În lecția 23, „O introducere în clasele C++", ați creat primele dumneavoastră clase în C++. Ați inclus atunci eticheta *public* în cadrul definiției de clasă pentru a oferi programului acces la fiecare dintre membrii clasei. În lecția de față, veți afla despre modul în care membrii *privați* și *publici* ai unei clase determină acei membri pe care programele îi pot accesa direct prin intermediul operatorului punct. Așa cum veți vedea, programele pot accesa membrii *publici* din cadrul oricărei funcții. Pe de altă parte, membrii *privați* pot fi accesați de programe numai cu ajutorul funcțiilor clasei. În acest fel, prin utilizarea membrilor *privați* în clase, un obiect poate controla modul în care un program îi utilizează membrii de date. Această lecție studiază în detaliu membrii *publici* și *privați*. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru a determina modul în care programele accesează membrii unei clase, C++ vă permite să definiți membrii ca *publici* sau *privați*.
- Membrii *privați* permit unei clase să ascundă acele informații ale clasei pe care programul nu are nevoie să le cunoască sau să le acceseze direct.
- Clasele care folosesc membri *privați* oferă *funcții de interfață* care accesează acei membri *privați*.

Așa cum menționam în lecția 23, la definirea unei clase ar trebui să plasați în cadrul definiției cât mai multe informații posibile despre obiectul corespunzător. În acest fel, obiectele devin autonome, ceea ce poate spori reutilizabilitatea lor în cadrul altor programe.

Despre ascunderea informațiilor

Așa cum ați învățat, o clasă conține date și metode (funcții). Pentru a utiliza o clasă, programele trebuie să cunoască informațiile pe care le păstrează acea clasă (membrii săi de date) și metodele care manipulează respectivele informații (funcțiile). Nu este nevoie ca un program să cunoască modul în care funcționează metodele. Pentru programe este să suficient să cunoască sarcinile îndeplinite de metode. De exemplu, să presupunem că avem o clasă *fișier*.

În mod ideal, programul nu trebuie să știe decât că această clasă oferă metodele *fisier.tipareste*, care tipărește o copie formatată a fișierului curent, și *fisier.sterge*, care șterge fișierul. Programul nu trebuie să știe cum anume funcționează aceste două metode. Cu alte cuvinte, programul ar trebui să trateze clasa ca pe o „cutie neagră". El știe ce metode să apeleze și ce parametrii să transmită acestor metode, dar nu cunoaște operațiile care se efectuează de fapt în interiorul clasei (cutia neagră).

Ascunderea informațiilor reprezintă procesul de punere la dispoziția programelor a unui minim de informații ale clasei de care programele au nevoie pentru utilizarea acelei clase. Membrii *privați* și *publici* din clasele C++ vă ajută la această ascundere a informațiilor în cadrul programelor. Toate clasele pe care le-ați creat în lecția 23 foloseau eticheta *public* pentru a face toți membrii de clasă *publici*, adică vizibili în întregul program. Din acest motiv, programul putea accesa direct oricare dintre membrii de clasă prin intermediul operatorului punct, ca mai jos:

```
class employe {
  public:
    char name[64];
    long employee_id;
    float salary;
    void show_employee(void);
}
```

Atunci când creați o clasă, este posibil să existe membri ai căror valori sunt folosite în interiorul clasei pentru a efectua diferite operații, dar pe care un program nu are de ce să le acceseze. Astfel de membri sunt *membri privați* și ei ar trebui ascunși programului. În mod implicit, dacă nu specificați eticheta *public*, C++ presupune că toți membrii clasei sunt *privați*. Programele nu pot accesa membrii *privați* ai unei clase prin intermediul operatorului punct. Membrii *privați* ai unei clase pot fi accesați numai de către funcțiile membru ale acelei clase. La crearea unei clase veți împărți membrii în *privați* și *publici*, așa cum se vede aici:

```
class o_clasa {
  public:
    int o_variabila;                              |
    void initializare_privati(int, float);        |—— Membri publici
    void afiseaza_date(void);                     |
  private:                  |
    int valoare_cheie;      |———————————————————————— Membri privați
    float numar_cheie;      |
}
```

După cum puteți observa, etichetele *public* și *private* vă permit stabilirea simplă a membrilor *privați* și *publici*. În acest exemplu, programul poate folosi operatorul punct pentru a accesa membrii *publici*, ca mai jos:

```
o_clasa obiect;      // Cream un obiect
obiect.o_variabila = 1001;
obiect.initializare_privati(2002, 1.2345);
obiect.afiseaza_date();
```

Dacă programul încearcă să acceseze membrii *privați valoare_cheie* sau *numar_cheie* cu ajutorul operatorului punct, compilatorul va genera erori de sintaxă.

Ca o regulă generală, membrii de date ai clasei sunt, de regulă, protejați de accesarea directă de către program prin fixarea lor ca *privați*. În acest fel, programele nu pot atribui valori membrilor direct prin operatorul punct. În schimb, programele vor trebui să apeleze o metodă a clasei pentru a atribui valori. Prin prevenirea accesului direct al programelor la membrii de date puteți asigura faptul că un program atribuie întotdeauna valori corecte membrilor de date ai unei clase. De exemplu, să presupunem că obiectul *reactor_nuclear* dintr-un program conține variabila numită *stare_termica*, a cărei valoare trebuie să fie întotdeauna între 1 și 5. Dacă membrul *stare_termica* este *public*, programul va putea accesa direct valoarea membrului, modificând-o în orice fel. De exemplu, instrucțiunea următoarea atribuie valoarea 101 (care se află în afara intervalului dintre 1 și 5) membrului *stare_termica* al clasei:

```
reactor_nuclear.stare_termica = 101;
```

Dacă fixați, în schimb, variabila ca fiind *privată*, puteți utiliza o metodă a clasei, precum *fixeaza_stare_termica*, pentru a atribui o valoare acestui membru. Așa cum se vede aici, funcția *fixeaza_stare_termica* poate testa valoarea pe care programul dorește să o atribuie membrului, asigurând astfel validitatea valorii:

```
int fuziune::fixeaza_stare_termica(int valoare)
  {
  if ((valoare > 0) && (valoare <= 5))
    {
      stare_termica = valoare;
      return(0);      // Atribuire reusita
    }
  else
    return(-1);      // Valoare incorecta
  }
```

Metodele unei clase care controlează accesul la membrii de date se numesc *funcții de interfață*. La crearea claselor veți proteja datele acestora cu ajutorul funcțiilor de interfață.

Despre membrii publici și privați

Clasele C++ conțin date și metode. Pentru a determina care membri pot fi accesați direct de programe prin intermediul operatorului punct, C++ vă permite definirea de membri *publici* și *privați*. Programele pot accesa direct orice membru *public* cu ajutorul operatorului punct. Pe de altă parte, membrii *privați* nu pot fi accesați decât de metodele clasei. Ca o regulă, ar trebui să protejați majoritatea membrilor de date ai unei clase prin stabilirea lor ca *privați*. Singura cale prin care programele vor putea apoi să atribuie o valoare unui membru de date va fi utilizarea unei funcții a clasei, aceasta putând să testeze și să valideze valoarea în cauză.

Utilizarea membrilor publici și privați

Programul următor, *InfoHide.CPP*, ilustrează modul de utilizare a membrilor *publici* și *privați*. Programul definește un obiect de tip *employee*, ca în continuare:

```
class employee {
  public:
    int assign_values(char *, long, float);
    void show_employee(void);
    int change_salary(float);
    long get_id(void);
  private:
    char name[64];
    long employee_id;
    float salary;
}
```

După cum se vede, clasa protejează toți membrii săi de date prin declararea acestora ca *privați*. Pentru accesarea unui membru de date, programul trebuie să apeleze la una dintre funcțiile de interfață *publice*. Iată implementarea programului *InfoHide.CPP*:

```
#include <iostream.h>
#include <string.h>

class employee {
  public:
    int assign_values(char *, long, float);
    void show_employee(void);
    int change_salary(float);
    long get_id(void);
```

```
  private:
    char name[64];
    long employee_id;
    float salary;
};

int employee::assign_values(char *emp_name,
  long emp_id, float emp_salary)
  {
    strcpy(name, emp_name);
    employee_id = emp_id;

    if (emp_salary < 50000.0)
    {
      salary = emp_salary;
      return(0); // Atribuire reusita
    }
    else
      return(-1); // Salariu eronat
  }

void employee::show_employee(void)
  {
    cout << "Employee: " << name << endl;
    cout << "Id: " << employee_id << endl;
    cout << "Salary: " << salary << endl;
  }

int employee::change_salary(float new_salary)
  {
    if (new_salary < 50000.0)
    {
      salary = new_salary;
      return(0); // Atribuire reusita
    }
    else
      return(-1); // Salariu eronat
  }
```

```
long employee::get_id(void)
  {
    return(employee_id);
  }

void main(void)
  {
    employee worker;
    if (worker.assign_values("Happy Jamsa", 101, 10101.0)
      == 0)
    {
      cout << "Employee values assigned" << endl;
      worker.show_employee();
      if (worker.change_salary(35000.00) == 0)
        {
          cout << "New salary assigned" << endl;
          worker.show_employee();
        }
    }
    else
      cout << "Invalid salary specified" << endl;
  }
```

Luaţi-vă puţin timp pentru a inspecta cu atenţie instrucţiunile programului. Deşi este un program lung, funcţiile sale sunt foarte explicite. Metoda *assign_values* iniţializează datele *private* ale clasei. Metoda foloseşte o instrucţiune *if* pentru a asigura o valoare corectă pentru salariu. Metoda *show_employee* afişează membrii de date *privaţi*. După compilarea şi rularea cu succes a programului *InfoHide.CPP*, editaţi programul şi încercaţi să accesaţi direct un membru de date *privat* prin utilizarea operatorului punct în cadrul programului principal. Pentru că membrii *privaţi* nu pot fi accesaţi direct, compilatorul va genera erori de sintaxă.

Despre funcţiile de interfaţă

Pentru a evita potenţiale erori, este bine să limitaţi accesul programelor la datele unei clase prin definirea membrilor de date ai clasei ca *privaţi*. În acest fel, un program nu poate accesa membrii de date ai clasei prin intermediul operatorului punct. În schimb, clasa va trebui să definească funcţii de interfaţă cu ajutorul cărora programul să poată atribui valori membrilor *privaţi*. Funcţiile de interfaţă, la rândul lor, vor testa şi valida valorile pe care programul încearcă să le atribuie.

Utilizarea operatorului de rezoluție globală pentru membrii de clasă

Dacă studiați funcțiile din programul *InfoHide.CPP*, veți vedea că numele de parametrii sunt de multe ori precedate de literele *emp_*, așa cum se vede aici:

```
int employee::assign_values(char *emp_name,
  long emp_id, float emp_salary)
```

Funcțiile utilizează literele *emp_* pentru a preveni conflictele dintre numele parametrilor și numele membrilor clasei. Atunci când apar astfel de conflicte de nume, o soluție posibilă este precedarea numelor de membri ai clasei cu numele clasei și operatorul de rezoluție globală (::). Funcția următoare specifică operatorul de rezoluție globală și numele clasei în fața numelor de membri ai clasei. În acest fel, oricine citește instrucțiunile știe care nume corespund clasei *employee* și care nume reprezintă parametrii, ca mai jos:

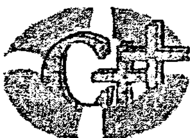
```
int employee::assign_values(char *name,
  long employee_id, float salary)
  {
    strcpy(employee::name, name);
    employee::employee_id = employee_id;
    if (salary < 50000.0)
    {
      employee::salary = salary;
      return(0); // Atribuire reusita
    }
    else
      return(-1); // Salariu eronat
  }
```

Atunci când creați funcții care lucrează cu membrii clasei, ar trebui să specificați în această manieră numele clasei și operatorul de rezoluție globală pentru a evita conflictele de nume.

Membrii privați nu sunt neapărat date

În exemplele prezentate de această lecție, membrii *privați* au fost de fiecare dată membrii de date. Pe măsură ce definițiile de clase cresc în complexitate, ați putea avea funcții utilizate de către celelalte metode ale clasei, dar care nu vreți să fie accesate direct de restul programului. În astfel de cazuri este suficient să declarați metoda respectivă ca membru *privat*. Dacă o funcție a unei clase nu este *publică*, programul nu poate apela acea funcție prin intermediul operatorului punct. În schimb, funcțiile membru *private* pot fi apelate numai de către alți membrii ai clasei.

Utilizarea operatorului global de rezoluție pentru specificarea membrilor de clasă



Atunci când scrieți funcții membre ale unei clase, este posibil ca numele unei variabile locale folosite să intre în conflict cu numele unui membru al clasei. În mod implicit, numele variabilei locale are prioritate în fața numelui de membru al clasei. Dacă apar astfel de conflicte de nume, funcția poate accesa membrul clasei specificând numele de clasă și operatorul de rezoluție globală, ca mai jos:

```
nume_clasa::nume_membru = o_valoare;
```

Ce trebuie să știți

Controlarea accesului programelor asupra membrilor de clasă duce la scăderea șanselor de apariție a erorilor care rezultă din utilizarea inadecvată în program a membrilor. Pentru a controla accesul la membrii unei clase veți apela la membrii *privați*. Cele mai multe definiții de clase C++ pe care le veți întâlni vor conține o combinație de membri *publici* și *privați*. Una dintre cele mai uzuale operații pe care programele le efectuează la crearea unui obiect este inițializarea membrilor de date ai obiectului. În lecția 25, „Despre funcțiile constructor și destructor", veți vedea că C++ vă permite definirea unei funcții speciale, numite *constructor*, pe care C++ o invocă automat de fiecare dată când creați un obiect. Prin intermediul funcției constructor, programele pot inițializa cu ușurință membrii unei clase. Dar înainte de a trece la lecția 25, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Membrii unei clase pot fi *publici* sau *privați*. Membrii *publici* pot fi accesați direct de programe prin intermediul operatorului punct. Pe de altă parte, membrii privați pot fi accesați numai prin intermediul metodelor clasei.
- ☑ Dacă nu se specifică altfel, C++ presupune că toți membrii sunt *privați*.
- ☑ Programele atribuie valori și accesează membrii *privați* prin intermediul funcțiilor de interfață.
- ☑ Atunci când scrieți programe care manipulează membrii de clasă, puteți să rezolvați eventualele conflicte de nume prin precedarea fiecărui nume de membru cu numele clasei și operatorul global de rezoluție (::), ca de pildă *employee::name*.

Lecția 25

Despre funcțiile constructor și destructor

Atunci când creați obiecte, una dintre cele mai uzuale operații pe care le veți efectua în programe va fi inițializarea membrilor de date ai obiectelor. Așa cum ați învățat în lecția 24, „Despre datele publice și private“, singurul fel în care programele pot accesa membri de date *privați* este prin utilizarea unei funcții a clasei. Pentru a simplifica procesul de inițializare a membrilor de date, C++ vă permite definirea unei funcții speciale *constructor*, specifică fiecărei clase, care va fi apelată la fiecare creare a unui obiect. În mod simetric, C++ permite și specificarea unei funcții *destructor* pe care o apelează la distrugerea unui obiect. Lecția de față discută detaliat despre funcțiile constructor și destructor. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Funcțiile constructor sunt metode ale claselor ce înlesnesc programelor inițializarea membrilor de date ai claselor.
- Funcțiile constructor au același nume cu clasa respectivă; numele funcției constructor nu se precede, însă, cu cuvântul cheie *void*.
- Funcțiile constructor nu întorc nici o valoare.
- De fiecare dată când un program creează o variabilă de tipul unei clase, C++ apelează funcția constructor, în cazul în care aceasta există.
- Rularea unui program implică alocarea de memorie în scopul stocării de informații ale diferitelor obiecte. La distrugerea unui obiect, C++ apelează o funcție specială destructor care poate elibera această memorie, făcând, dacă vreți, curățenie în urma obiectului.
- Funcțiile destructor au același nume cu clasa respectivă, dar acest nume trebuie precedat de caracterul tildă (~).
- Funcțiile destructor nu întorc nici o valoare. Asemeni funcțiilor constructor, numele unei funcții destructor nu sunt precedate de cuvântul cheie *void*.

Nu vă lăsați intimidat de termenii constructor și destructor. Gândiți-vă, mai degrabă, că o funcție constructor este o funcție care vă ajută să asamblați (construiți) un obiect. Analog, o funcție destructor este o funcție care vă ajută la distrugerea unui obiect. Utilizarea funcțiilor destructor este foarte uzuală în cazurile în care un obiect alocă memorie, iar înainte ca programul să distrugă obiectul doriți ca acea memorie să fie la rândul său eliberată.

Crearea unei funcții constructor simple

O *funcție constructor* este o metodă a unei clase care are același nume cu clasa respectivă. De exemplu, în cazul unei clase numite *employee*, numele funcției constructor este de

asemenea *employee*. Similar, pentru o clasă numită *dog*, numele funcției constructor este *dog*. Dacă programul definește o funcție constructor, C++ va apela automat această funcție de fiecare dată când creați un obiect de tipul clasei respective. Programul următor, *ConStruc.CPP*, creează o clasă numită *employee*. Programul definește, de asemenea, o funcție constructor numită *employee* care atribuie valori inițiale pentru obiect. O funcție constructor nu poate întoarce nici o valoare; cu toate acestea, funcția nu se declară ca *void*. În schimb, nu veți specifica nici un tip întors, așa cum se vede aici:

```
class employee {
  public:
    employee(char *, long, float); // Functia constructor
    void show_employee(void);
    int change_salary(float);
    long get_id(void);
  private:
    char name[64];
    long employee_id;
    float salary;
};
```

În cadrul programului, funcția constructor se definește ca orice altă metodă a clasei, după cum puteți vedea:

```
employee::employee(char *name, long employee_id, float salary)
  {
    strcpy(employee::name, name);
    employee::employee_id = employee_id;
    if (salary < 50000.0)
      employee::salary = salary;
    else
      // Salariul specificat este eronat
      employee::salary = 0.0;
  }
```

Așa cum se observă, funcția constructor nu întoarce către apelant nici o valoare. De asemenea, nu este precizat nici tipul *void*. În acest exemplu, funcția specifică operatorul de rezoluție globală și numele clasei înaintea fiecărui membru. Programul *ConStruc.CPP* este prezentat în continuare:

```
#include <iostream.h>
#include <string.h>

class employee {
  public:
    employee(char *, long, float);
    void show_employee(void);
    int change_salary(float);
    long get_id(void);
  private:
    char name[64];
    long employee_id;
    float salary;
};

employee::employee(char *name, long employee_id, float salary)
  {
    strcpy(employee::name, name);
    employee::employee_id = employee_id;
    if (salary < 50000.0)
      employee::salary = salary;
    else
      // Salariul specificat este eronat
      employee::salary = 0.0;
  }

void employee::show_employee(void)
  {
    cout << "Employee:   << name << endl;
    cout << "Id: " << employee_id << endl;
    cout << "Salary: " << salary << endl;
  }

void main(void)
  {
    employee worker("Happy Jamsa", 101, 10101.0);

    worker.show employee();
  }
```

Observați că programul *ConStruc.CPP* specifică în declarația obiectului *worker* valorile inițiale ale acestuia, plasate între paranteze, asemănător unui apel de funcție. Atunci când folosește funcții constructor, un program poate transmite parametri constructorului la declararea obiectului, ca mai jos:

```
employee worker("Happy Jamsa", 101, 10101.0);
```

Dacă programul ar fi creat mai multe obiecte *employee*, fiecare dintre acestea ar fi putut fi inițializat prin intermediul constructorului, așa cum se ilustrează aici:

```
employee worker("Happy Jamsa", 101, 10101.0);
employee Secretary("John Doe", 57, 20000.0);
employee manager("Jane Doe", 1022, 30000.0);
```

Despre funcțiile constructor



O funcție constructor este o funcție specială pe care C++ o apelează automat de fiecare dată când creați un obiect. Modul cel mai uzual de utilizare a constructorilor îl reprezintă inițializarea membrilor de date ai unui obiect. Funcțiile constructor au același nume cu clasa obiectului corespunzător. O clasă numită *fișier*, de pildă, utilizează un constructor cu numele *fișier*. Funcțiile constructor se definesc în program asemeni oricărei metode a clasei. Singura diferență este că pentru funcțiile constructor nu se specifică un tip întors. La declararea ulterioară a unui obiect puteți transmite parametri constructorului sub forma următoare:

```
nume_clasa obiect(valoare1, valoare2, valoare3)
```

Specificarea valorilor implicite pentru parametrii funcțiilor constructor

Așa cum ați văzut în lecția 16, „Precizarea valorilor implicite pentru parametri“, C++ permite specificarea de valori implicite pentru parametri. Dacă utilizatorul nu specifică valorile tuturor parametrilor, funcția va utiliza valorile implicite. Funcțiile constructor nu fac nici ele excepție; programele pot specifica valori implicite ca în cazul oricărei funcții. De exemplu, funcția constructor *employee* care urmează, folosește valoarea implicită 10000,00 pentru salariu, în cazul în care programul nu precizează o altă valoare la crearea obiectului. Programul trebuie, totuși, să precizeze un nume de angajat și un număr de identificare:

```
employee::employee(char *name, long employee id,
  float salary = 10000.00)
  {
     strcpy(employee::name, name);
     employee::employee_id = employee_id;
     if (salary < 50000.0)
        employee::salary = salary;
     else
        // Salariul specificat este eronat
        employee::salary = 0.0;
  }
```

Supradefinirea funcțiilor constructor

După cum ați învățat în lecția 15 „Supradefinirea funcțiilor", C++ permite programelor supradefinirea funcțiilor prin specificarea unor funcții alternative ce corespund unor tipuri diferite de parametri. Astfel, C++ vă permite să supradefiniți funcțiile constructor. Programul următor, *ConsOver.CPP*, supradefinește funcția constructor *employee*. Prima definiție a funcției constructor solicită programului specificarea unui nume de angajat, al unui număr de identificare și al unui salariu. A doua definiție a funcției constructor solicită utilizatorului introducerea unui salariu în cazul în care programul nu precizează unul, așa cum se vede aici:

```
employee::employee(char *name, long employee_id)
  {
     strcpy(employee::name, name);
     employee::employee_id = employee_id;
     do {
        cout << ""Enter a salary for " << name <<
           " less than $50,000: ";
        cin >> employee::salary;
     } while (salary >= 50000.0);
}
```

În interiorul definiției clasei trebuie specificate ambele prototipuri de funcții, ca mai jos:

```
class employee {
  public:
    employee(char *, long, float);     Prototipurile funcției
    employee(char *, long);            supradefinite
    void show_employee(void);
    int change_salary(float);
    long get_id(void);
  private:
    char name[64];
    long employee_id;
    float salary;
};
```

Programul *ConsOver.CPP* este următorul:

```
#include <iostream.h>
#include <string.h>

class employee {
  public:
    employee(char *, long, float);
    employee(char *, long);
    void show_employee(void);
    int change_salary(float);
    long get_id(void);
  private:
    char name[64];
    long employee_id;
    float salary;
};

employee::employee(char *name, long employee_id,
  float salary)
  {
    strcpy(employee::name, name);
    employee::employee_id = employee_id;
    if (salary < 50000.0)
      employee::salary = salary;
```

```
        else
          // Salariul specificat este eronat
          employee::salary = 0.0;
     }

   employee::employee(char *name, long employee_id)
     {
        strcpy(employee::name, name);
        employee::employee_id = employee_id;
        do {
          cout << "Enter a salary for " << name <<
            " less than $50,000 ";
          cin >> employee::salary;
        } while (salary >= 50000.0);
     }

   void employee::show_employee(void)
     {
        cout << "Employee: " << name << endl;
        cout << "Id: " << employee_id << endl;
        cout << "Salary: " << salary << endl;
     }

   void main(void)
     {
        employee worker("Happy Jamsa", 101, 10101.0);
        employee manager("Jane Doe", 102);
        worker.show_employee();
        manager.show_employee();
     }
```

La compilarea și rularea programului *ConsOver.CPP* veți fi solicitat să introduceți un salariu pentru Jane Doe. După precizarea unei sume, execuția programului continuă prin afișarea de informații despre ambii angajați.

Despre funcțiile destructor

Așa cum vă permite definirea unei funcții constructor care este apelată automat la crearea unui obiect al unei clase, C++ vă permite totodată definirea unei *funcții destructor* care este apelată la distrugerea obiectului. În lecțiile următoare veți învăța să creați liste de obiecte care cresc sau se micșorează pe parcursul rulării programului. Pentru a crea astfel de liste dinamice, de exemplu, programul va aloca dinamic memoria necesară pentru a reține obiectele (o operație pe care nu ați învățat încă să o efectuați). Într-o astfel de situație veți putea să creați și să distrugeți obiecte pe parcursul execuției programului. Utilizarea unei funcții destructor capătă astfel sens.

Toate programele pe care le-ați scris până acum au creat obiecte de la bun început, prin declararea acestora. La terminarea acestor programe, C++ distruge obiectele respective. În cazul în care definiți o funcție destructor, C++ va apela automat destructorul fiecărui obiect odată cu încheierea programului (atunci când C++ distruge obiectele). Asemeni constructorilor, funcțiile destructor au același nume cu clasa corespunzătoare. Acest nume este precedat, însă, de un caracter tildă (~), ca mai jos:

```
~nume_clasa(void)
  {
    // Instructiunile functiei
  }
```

~ Indică un destructor

Spre deosebire de constructori, funcțiile destructor nu pot primi parametri.

Programul următor, *Destruct.CPP*, definește destructorul clasei *employee* astfel:

```
employee::~employee(void)
  {
    cout << "Destroying the object for  << name << endl;
  }
```

În exemplul de față, funcția destructor se rezumă la a afișa pe ecran un mesaj ce anunță distrugerea obiectului de către C++. La încheierea programului, C++ apelează automat această funcție pentru fiecare obiect. Programul *Destruct.CPP* este prezentat în cele ce urmează:

```
#include <iostream.h>
#include <string.h>
class employee {
  public:
    employee(char *, long, float);
    ~employee(void);
```

```
      void show_employee(void);
      int change_salary(float);
      long get_id(void);
   private:
      char name[64];
      long employee_id;
      float salary;
};

employee::employee(char *name, long employee_id,
   float salary)
   {
      strcpy(employee::name, name);
      employee::employee_id = employee_id;
      if (salary < 50000.0)
         employee::salary = salary;
      else
         // Salariul specificat este eronat
         employee::salary = 0.0;
   }

employee::~employee(void)
   {
      cout << "Destroying the object for " << name << endl;
   }

void employee::show_employee(void)
   {
      cout << "Employee: " << name << endl;
      cout << "Id: " << employee_id << endl;
      cout << "Salary: " << salary << endl;
   }

void main(void)
   {
      employee worker("Happy Jamsa", 101, 10101.0);

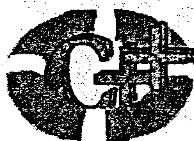
      worker.show_employee();
   }
```

După compilarea și rularea programului *Destruct.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> Destruct <Enter>
Employee: Happy Jamsa
Id: 101
Salary: 10101
Destroying the object for Happy Jamsa
```

După cum puteți vedea, programul apelează automat funcția destructor, fără a necesita un apel explicit de funcție. Deocamdată, este probabil ca programele dumneavoastră să nu utilizeze funcțiile destructor. Dar atunci când programele vor ajunge să aloce memorie în cadrul obiectelor, veți vedea că funcțiile destructor reprezintă o soluție elegantă pentru eliberarea memoriei odată cu distrugerea de către program a obiectelor.

Despre funcțiile destructor



O funcție destructor este o funcție pe care C++ o execută automat atunci când un obiect este distrus de către compilator sau de către program. Funcțiile destructor au același nume cu cel al claselor corespunzătoare; numele funcției destructor trebuie, însă, precedat de un caracter tildă (~), ca de pildă *~employee*. Definirea în program a metodelor destructor se face ca în cazul oricăror alte metode ale clasei.

Ce trebuie să știți

Constructorii și destructorii sunt funcții speciale ale claselor pe care C++ le apelează automat atunci când programul creează sau distruge un obiect. Majoritatea programelor folosesc funcțiile constructor pentru inițializarea membrilor de date ai unei clase. Programele simple pe care le creați acum nu necesită, probabil, utilizarea de funcții destructor. În lecția 26, „Despre supradefinirea operatorilor", veți vedea cum se pot supradefini operatorii. Cu alte cuvinte, ați putea redefini simbolul plus (+) astfel încât să aibă ca efect concatenarea a două șiruri. Așa cum ați văzut, un tip de date (precum *char*, *float* și *int*) definește o mulțime de valori ce pot fi reținute într-o variabilă și o mulțime de operații pe care programele le pot efectua asupra variabilei. Definirea unei clase echivalează, în fond, cu definirea unui tip. C++ vă permite să specificați comportamentul operatorilor în cazul unui astfel de tip. Dar înainte de a trece la lecția 26, asigurați-vă că ați reținut următoarele aspecte importante:

☑ O funcție constructor este o funcție specială pe care C++ o apelează automat atunci când programul creează un obiect. Funcția constructor are același nume cu clasa obiectului corespunzător.

☑ Funcțiile constructor nu întorc nici o valoare, dar nici nu se definesc ca având tipul *void*. Pur și simplu nu se specifică nici un tip al valorii întoarse.

- ☑ Atunci când creează un obiect, programele pot transmite parametri către funcţia constructor în cadrul declaraţiei obiectului.
- ☑ C++ vă permite supradefinirea funcţiilor constructor şi specificarea de valori implicite pentru parametrii acestora.
- ☑ O funcţie destructor este o funcţie specială pe care C++ o apelează automat de fiecare dată când programul creează sau distruge un obiect. Funcţia destructor are, de asemenea, acelaşi nume cu cel al clasei obiectului corespunzător, însă acest nume este precedat de un caracter tildă (~).

Lecția 26

Despre supradefinirea operatorilor

Așa cum ați văzut, tipul unei variabile indică o mulțime de valori ce pot fi păstrate de variabilă și o mulțime de operații pe care le puteți efectua asupra variabilei respective. La utilizarea unei variabile de tip *int*, de pildă, programul poate să adune, să scadă, să înmulțească și să împartă valori. În cazul unui șir de caractere, în schimb, folosirea operatorului plus pentru adunarea a două șiruri nu are nici un sens. Definirea unei clase în cadrul programului înseamnă, practic, definirea unui nou tip. Din acest motiv, C++ vă permite să specificați operațiile corespunzătoare acestui nou tip. *Supradefinirea operatorilor* este procesul de modificare a semnificației unui operator (așa cum este operatorul plus (+), pe care C++ îl folosește în mod normal pentru adunare) în scopul utilizării acestuia de către o anumită clasă. În lecția de față veți defini o clasă *string* și veți supradefini operatorii plus și minus. În cazul obiectelor *string*, operatorul plus va atașa caracterele specificate la conținutul curent al șirului. În mod similar, operatorul minus va elimina fiecare apariție din șir a unei litere specificate. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Supradefinirea operatorilor are ca scop sporirea lizibilității programelor. Cu toate acestea, supradefinirea operatorilor este recomandată numai atunci când aceasta face ca programul să devină mai ușor de înțeles.
- Pentru supradefinirea unui operator, programele apelează la cuvântul cheie *operator* din C++.
- Supradefinirea unui operator presupune specificarea unei funcții pe care C++ o apelează de fiecare dată când clasa utilizează operatorul supradefinit. Funcția este cea care va efectua operația corespunzătoare.
- Atunci când un program supradefinește un operator pentru o anumită clasă, semnificația operatorului este alterată numai în cazul acelei clase – restul programului va utiliza în continuare acel operator pentru a efectua operația obișnuită.
- C++ permite programelor să supradefinească cea mai mare parte a operatorilor; există, totuși, patru operatori (vedeți tabelul 26) pe care programele nu îi pot supradefini.

Supradefinirea operatorilor poate să simplifice operațiile elementare cu clase și să sporească lizibilitatea programelor. Experimentați cu fiecare program prezentat în această lecție și veți vedea că supradefinirea operatorilor este destul de simplă.

Supradefinirea operatorilor plus și minus

Atunci când supradefiniți un operator în cadrul unei clase, semnificația acelui operator rămâne neschimbată în cazul celorlalte tipuri de variabile. De exemplu, atunci când supradefiniți operatorul plus pentru clasa *string*, semnificația acestuia va rămâne neschimbată la utilizarea sa pentru adunarea a două numere. Atunci când întâlnește un operator în program, compilatorul de C++ determină operația care trebuie efectuată în funcție de tipul variabilei corespunzătoare. Următoarea definiție de clasă duce la crearea clasei *string*. Clasa conține un singur membru de date, și anume șirul de caractere propriu-zis. În clasă se află mai multe metode și nu este definit nici un operator, așa cum se vede aici:

```
class string {
  public:
    string(char *); // Constructor
    void str_append(char *);
    void chr_minus(char);
    void show_string(void);
  private:
    char data[256];
};
```

După cum puteți observa, definiția clasei conține funcția *str_append*, care atașează caracterele specificate la conținutul șirului din clasă. Similar, funcția *chr_minus* elimină fiecare apariție a caracterului specificat din șirul clasei. Programul următor, *StrClass.CPP*, utilizează clasa *string* pentru a crea și manipula două obiecte șir de caractere:

```
#include <iostream.h>
#include <string.h>

class string {
  public:
    string(char *)  // Constructor
    void str_append(char *);
    void chr_minus(char);
    void show_string(void);
  private:
    char data[256];
};

string::string(char *str)
```

```
   {
     strcpy(data, str);
   }

void string::str_append(char *str)
   {
     strcat(data, str);
   }

void string::chr_minus(char letter)
   {
     char temp[256];
     int i, j;

     for (i = 0, j = 0; data[i]; i++)
        // Litera trebuie eliminata?
        if (data[i] != letter)
           // Daca nu, o atasam la temp
           temp[j++] = data[i];

     temp[j] = NULL; // Sfarsitul sirului temp

     // Copiem continutul lui temp inapoi in sirul data
     strcpy(data, temp);
   }

void string::show_string(void)
   {
     cout << data << endl;
   }

void main(void)
   {
     string title("Rescued By C++");
     string lesson("Understanding Operator Overloading");

     title.show_string();
     title.str_append(" rescued me!");
     title.show_string();
```

```
      lesson.show_string();
      lesson.chr_minus('n');
      lesson.show_string();
   }
```

Precum vedeți, programul utilizează funcția *str_append* pentru a atașa caractere la variabila șir *title*. De asemenea, programul folosește funcția *chr_minus* pentru a elimina fiecare apariție a literei *n* din șirul de caractere *lesson*. În exemplul de față, programul *StrClass.CPP* efectuează aceste operații prin intermediul unor apeluri de funcții. Prin supradefinirea operatorilor, însă, programul poate efectua operații identice cu ajutorul operatorilor plus (+) și minus (-).

Supradefinirea unui operator se face prin specificarea în cadrul prototipului și definiției unei funcții a cuvântului cheie *operator* din C++, informând astfel compilatorul de C++ că respectiva clasă va utiliza metoda în cauză ca operator. Următoarea definiție de clasă, de exemplu, folosește cuvântul cheie *operator* pentru a asocia operatorii plus și minus funcțiilor *str_append* și *chr_minus* din cadrul clasei *string*:

```
class string {
  public:
    string(char *); // Constructor
    void operator +(char *);     |
    void operator -(char);       |—— Definirea operatorilor clasei
    void show_string(void);
  private:
    char data[256];
};
```

Așa cum se vede, clasa supradefinește operatorii plus și minus. După cum am menționat, la supradefinirea unui operator în cadrul unei clase, acea clasă trebuie să specifice o funcție care implementează operația corespunzătoare respectivului operator. În cazul operatorului plus, definiția funcției devine următoarea:

```
void string::operator +(char *str)
  {
    strcat(data, str);
  }
```

După cum vedeți, definiția funcției nu mai specifică un nume, ci operatorul supradefinit de către clasă. Pentru supradefinirea operatorului plus, programul nu a modificat operațiile care sunt efectuate în interiorul funcției (codul acestei funcții este identic cu cel din funcția

str_append de mai devreme). În schimb, programul a înlocuit pur și simplu numele funcției cu cuvântul cheie *operator* și cu operatorul corespunzător. Programul următor, *OpOverld.CPP*, ilustrează utilizarea operatorilor supradefiniți plus și minus:

```
#include <iostream.h>
#include <string.h>

class string {
  public:
    string(char *); // Constructor
    void operator +(char *);
    void operator -(char);
    void show_string(void);
  private:
    char data[256];
};

string::string(char *str)
  {
    strcpy(data, str);
  }

void string::operator +(char *str)
  {
    strcat(data, str);
  }

void string::operator -(char letter)
  {
    char temp[256];
    int i, j;

    for (i = 0, j = 0; data[i]; i++)
      if (data[i] != letter)
        temp[j++] = data[i];

    temp[j] = NULL;

    strcpy(data, temp);
  }
```

```
void string::show_string(void)
  {
    cout << data << endl;
  }

void main(void)
  {
    string title("Rescued By C++");
    string lesson("Understanding Operator Overloading");

    title.show_string();
    title + " rescued me!";
    title.show_string();

    lesson.show_string();
    lesson - 'n';
    lesson.show_string();
  }
```

Puteți observa cum programul *OpOverld.CPP* utilizează operatorii supradefiniți ca aici:

```
// Atasam sirul "rescued me!"
title + " rescued me!";
lesson - 'n';      // Eliminam litera 'n'
```

În exemplul nostru, sintaxa operatorului este corectă, dar puțin ciudată. În mod normal, programele utilizează operatorul plus în cadrul unei expresii care întoarce un rezultat, așa cum este instrucțiunea *un_sir = titlu + "text";*. Atunci când definiți operatori proprii, C++ vă oferă relativ puțină libertate în a determina comportamentul acestora. Dar scopul supradefinirii operatorilor era, după cum vă amintiți, scrierea unor programe mai ușor de înțeles. Spre exemplu, programul următor, *Str_Over.CPP*, modifică puțin programul anterior pentru a permite efectuarea de operații asupra variabilelor *string* sub o formă ceva mai asemănătoare cu operațiile de atribuire uzuale:

```
#include <iostream.h>
#include <string.h>

class string {
  public:
    string(char *); // Constructor
```

```
    char * operator +(char *);
    char * operator -(char);
    void show_string(void)
  private:
    char data[256];
};
string::string(char *str)
  {
    strcpy(data, str);
  }
char * string::operator +(char *str)
  {
    return(strcat(data, str));
  }
char * string::operator -(char letter)
  {
    char temp[256];
    int i, j;
    for (i = 0, j = 0; data[i]; i++)
      if (data[i] != letter)
         temp[j++] = data[i];
    temp[j] = NULL;
    return(strcpy(data, temp));
  }
void string::show_string(void)
  {
    cout << data << endl;
  }
void main(void)
  {
    string title("Rescued By C++")
    string lesson("Understanding Operator Overloading")
    title.show_string();
```

```
        title = title + " rescued me!";
        title.show_string();

        lesson.show_string();
        lesson = lesson - 'n';
        lesson.show_string();
    }
```

Prin modificarea operatorilor supradefiniţi plus şi minus astfel încât aceştia să întoarcă pointeri la şiruri de caractere, programul poate utiliza acum aceşti operatori folosind sintaxa unei instrucţiuni de atribuire, ca aici:

```
title = title + " rescued me!";
lesson = lesson - 'n';
```

Dacă inspectaţi atent codul, veţi vedea că instrucţiunile de mai sus atribuie un pointer la un vector de caractere unui obiect *string*. Atunci când întâlneşte această atribuire, compilatorul de C++ apelează funcţia constructor a clasei *string*, iar aceasta atribuie conţinutul şirului de caractere membrului de date al obiectului.

Un alt exemplu

Atunci când creaţi propriile tipuri de date prin intermediul claselor, una dintre operaţiile uzuale pe care le veţi efectua va fi să verificaţi dacă două obiecte sunt identice. Apelând la supradefinirea operatorilor, un program poate să supradefinească operatorii relaţionali, inclusiv operatorii egal (==) şi diferit (!=). Următorul program, *Comp_Str.CPP*, adaugă clasei *string* un nou operator care testează dacă două obiecte *string* sunt egale. Cu ajutorul operatorului supradefinit, programele pot testa dacă două obiecte şir au acelaşi conţinut sub forma:

```
if (un_sir == alt_sir)
```

Programul *Comp_Str.CPP* este următorul:

```
#include <iostream.h>
#include <string.h>

class string {
  public:
    string(char *); // Constructor
    char * operator +(char *);
```

```
    char * operator -(char);
    int operator ==(string);
    void show_string(void);
  private:
    char data[256];
};

string::string(char *str)
  {
    strcpy(data, str);
  }

char * string::operator +(char *str)
  {
    return(strcat(data, str));
  }

char * string::operator -(char letter)
  {
    char temp[256];
    int i, j;

    for (i = 0, j = 0; data[i]; i++)
       if (data[i] != letter)
          temp[j++] = data[i];

    temp[j] = NULL;

    return(strcpy(data, temp));
  }

int string::operator ==(string str)
  {
    int i;

    for (i = 0; data[i] == str.data[i]; i++)
       if ((data[i] == NULL) && (str.data[i] == NULL))
          return(1); // Egal

    return(0); // Diferit
  }
```

```
void string::show_string(void)
  {
    cout << data << endl;
  }

void main(void)
  {
    string title("Rescued By C++");
    string lesson("Understanding Operator Overloading");
    string str("Rescued By C++");

    if (title == lesson)
      cout << "title and lesson are equal" << endl;

    if (str == lesson)
      cout << "str and lesson are equal" << endl;

    if (title == str)
      cout << "title and str are equal" << endl;
  }
```

După cum puteți vedea, o astfel de supradefinire a operatorilor poate face programele mai ușor de înțeles.

Operatori ce nu pot fi supradefiniți

În general, programele pot supradefini aproape orice operator din C++. Tabelul 26 conține acei operatori pe care C++ nu permite să-i supradefiniți în programe.

Operator	Scop	Exemplu
	Operatorul de membru al clasei	obiect.membru
	Operatorul de pointer la membru	obiect.*membru
	Operatorul de rezoluție globală	nume_clasa::membru
?:	Operatorul expresie condițională	c = (a > b) ? a : b;

Tabelul 26 *Operatorii din C++ pe care programele nu îi pot supradefini.*

Ce trebuie să știți

Supradefinirea operatorilor se referă la posibilitatea de a atribui o nouă semnificație unui operator atunci când o clasă anume utilizează acel operator. Prin supradefinirea operatorilor puteți să sporiți lizibilitatea programelor și să înlesniți înțelegerea acestora datorită exprimării operațiilor cu clase sub o formă mai sugestivă. În lecția 27, „Funcții și date

membru statice", veți vedea cum puteți să partajați date între obiecte cu ajutorul membrilor *statici* sau să apelați o metodă de clasă atunci când programul nu a declarat încă obiecte ale acelei clase. Dar înainte de a trece la lecția 27, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Supradefinirea unui operator presupune definirea clasei cu care va fi asociat respectivul operator.

☑ La supradefinirea unui operator, sensul noii definiții este valabil numai pentru o anumită clasă. Atunci când programul utilizează operatorul împreună cu variabile al căror tip este altul decât clasa în cauză (așa cum sunt variabilele de tip *int* și *float*), C++ folosește definiția originală a operatorului.

☑ Supradefinirea unui operator se face prin utilizarea cuvântului cheie *operator* pentru definirea acelei metode a unei clase pe care C++ o va apela ori de câte ori o variabilă de tipul clasei respective folosește operatorul în cauză.

☑ C++ nu permite programelor să supradefinească operatorul de membru (.), operatorul de pointer la membru (.*), operatorul de rezoluție (::) sau operatorul de expresie condițională (?:).

Lecția 27
Funcții și date membru statice

Până în acest moment, toate obiectele create în programe au dispus de propria mulțime de membri de date. În funcție de scopul programului, ar putea fi situații în care să doriți ca obiectele unei clase să partajeze una sau mai multe variabile. Spre exemplu, să presupunem că scrieți un program de salarii care gestionează orele de muncă pentru 1000 de angajați. Pentru a determina impozitul de plătit, programul trebuie să cunoască rata impozitului pentru fiecare angajat. Așadar, unul din membrii clasei este *rata_impozit*. Dacă rata impozitului este, însă, aceeași pentru toți angajații, programul ar putea partaja această informație pentru toate obiectele angajat.

În acest fel, programul reduce cantitatea de memorie utilizată prin eliminarea a 999 de copii ale aceleiași informații. Partajarea unui membru de clasă se face prin declararea respectivului membru ca *static*. Lecția de față prezintă pașii care trebuie urmați pentru partajarea de către obiecte a membrilor de clasă. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- C++ permite ca obiectele aceleiași clase să partajeze unul sau mai mulți membri ai clasei.
- Atunci când un program atribuie o valoare unui membru partajat, toate obiectele clasei respective au acces instantaneu la noua valoare.
- Pentru a crea un membru de date partajat în cadrul unei clase, numele acelui membru trebuie precedat de cuvântul cheie *static*.
- După declararea unui membru de clasă ca *static*, programul trebuie să declare o variabilă globală (în exteriorul definiției de clasă) care corespunde membrului partajat al clasei.
- Programele pot utiliza cuvântul cheie *static* pentru a face ca o metodă a unei clase să fie apelabilă chiar și atunci când programul nu a definit încă nici un obiect aparținând clasei respective.

Partajarea unui membru de date

Atunci când creați obiecte aparținând unei clase, fiecare obiect posedă, în mod normal, propria mulțime de membri de date. Ar putea exista, însă, situații în care obiectele aceleiași clase trebuie să partajeze unul sau mai mulți membri de date. În asemenea cazuri, declarați membrii de date respectivi ca *publici* sau *privați* și precedați tipul acestora cu cuvântul cheie *static*, așa cum se vede aici:

```
private:
    static int valoare_partajata;
```

După declararea clasei trebuie să definiți membrul respectiv ca variabilă globală, în exteriorul clasei, ca mai jos:

```
int nume_clasa::valoare_partajata;
```

Programul următor, *Share_It.CPP*, definește clasa *book_series* în care membrul *page_count* este partajat, fiind același pentru toate obiectele (cărți) ale clasei (colecție). Dacă programul modifică valoarea acestui membru, toate obiectele clasei vor resimți imediat modificarea, după cum se vede în continuare:

```
#include <iostream.h>
#include <string.h>
class book_series {
  public:
    book_series(char *, char *  float);
    void show_book(void);
    void set_pages(int);
  private:
    static int page_count;
    char title[64];
    char author[64];
    float price;
};
int book_series::page_count;
void book_series::set pages(int pages)
  {
    page_count = pages;
  }
book_series::book_series(char *title,
  char *author, float price)
  {
    strcpy(book_series::title, title);
```

```
      strcpy(book_series::author, author)
      book_series::price = price;
   }

void book_series::show_book(void)
   {
      cout << "Title: " << title << endl;
      cout << "Author:   << author << endl;
      cout << "Price:   << price << endl;
      cout << "Pages. " << page_count << endl;
   }

void main(void)
   {
      book_series programming("Rescued by C++, Third Edition",
         "Jamsa", 29.95);
      book_series upgrading("Rescued by Upgrading Your PC",
         "Jamsa", 24.95);

      word.set_pages(256)

      programming.show_book();
      upgrade.show_book()

      cout << endl << "Changing page count " << endl;

      programming.set_pages(512)

      programming.show_book()
      upgrade.show_book()
   }
```

Precum vedeți, clasa declară membrul *page_count* ca *static int.* Imediat după definiția clasei, programul declară membrul *page_count* ca variabilă globală. Atunci când programul modifică membrul *page_count*, schimbarea afectează instantaneu toate obiectele clasei *book_series.*

Partajarea membrilor unei clase



În funcție de program, pot exista situații în care trebuie să partajați anumite informații pentru toate obiectele unei clase. În acest scop, declarați membrul vizat ca *static*. Apoi, declarați membrul în afara clasei ca variabilă globală. Ulterior, toate obiectele acelei clase vor sesiza imediat orice modificare pe care programul o aduce membrului respectiv.

Utilizarea membrilor statici publici atunci când nu există obiecte

După cum ați aflat, un membru al clasei declarat ca *static* este partajat de toate obiectele ce aparțin acelei clase. Ar putea fi, totuși, cazuri în care programele nu au creat nici un obiect, dar au nevoie să utilizeze un astfel de membru. De exemplu, programul următor, *Use_Mbr.CPP*, folosește membrul *page_count* al clasei *book_series*, deși nu există nici un obiect al acestei clase:

```
#include <iostream.h>
#include <string.h>

class book_series {
  public:
    static int page_count;
  private:
    char title[64];
    char author[64];
    float price;
};

int book_series::page_count;

void main(void)
  {
    book_series::page_count = 256;
    cout << "The current page count is " <<
      book_series::page_count << endl;
  }
```

În exemplul prezentat, deoarece membrul *page_count* este definit în interiorul clasei ca *public*, programul poate accesa acest membru chiar în situația în care nu există nici un obiect al clasei *book_series*.

Utilizarea funcțiilor membru statice

Programele anterioare au ilustrat utilizarea membrilor de date statici. Într-un mod similar, C++ vă permite să definiți funcții membru (metode) ca fiind *statice*. Atunci când creați o metodă *statică*, programul poate apela funcția chiar și dacă nu există nici un obiect. Dacă, de pildă, o clasă conține o metodă pe care programul ar putea să o folosească pentru a manipula date exterioare clasei, ați putea declara acea metodă ca *statică*. Spre exemplu, clasa *menu* de mai jos utilizează o secvență specială de coduri pentru driverul ANSI în scopul ștergerii ecranului. Dacă în calculator este instalat driverul de dispozitiv *ANSI.SYS*, atunci metoda *clear_screen* poate fi folosită pentru ștergerea ecranului. Deoarece metoda este declarată ca *statică*, programul o poate apela chiar dacă nu există obiecte de tipul *menu*. Următorul program, *Clr_Scr.CPP*, folosește metoda *clear_screen* pentru a șterge ecranul:

```
#include <iostream.h>

class menu {
  public:
     static void clear_screen(void);
     // Aici s-ar afla alte metode
  private:
     int number_of_menu_options;
};

void menu::clear_screen(void)
  {
     cout << '\033' << "[2J";
  }

void main(void)
  {
     menu::clear_screen();
  }
```

Deoarece membrul *clear_screen* este declarat ca static, programul poate folosi această funcție pentru a șterge ecranul, chiar dacă nu există nici un obiect de tipul *menu*. Funcția *clear_screen* apelează la secvența de coduri ANSI *Esc[2J* pentru a șterge ecranul.

Utilizarea metodelor de clase în programe



La crearea metodelor de clase, ar putea apărea cazuri în care o funcție creată pentru utilizare în cadrul unei clase se dovedește utilă pentru operații din program care nu implică obiectele clasei respective. Spre exemplu, clasa *menu* de mai devreme definește o funcție *clear_screen* pe care ați putea să o utilizați în program. Atunci când o clasă conține o metodă pe care ați putea dori să o utilizați independent de un obiect al acelei clase, precedați prototipul metodei cu cuvântul cheie *static* și declarați respectiva metodă ca *publică*, așa cum se vede aici:

```
public:
   static void clear_screen(void);
```

Pentru apelarea funcției din program, utilizați operatorul de rezoluție globală, ca mai jos:

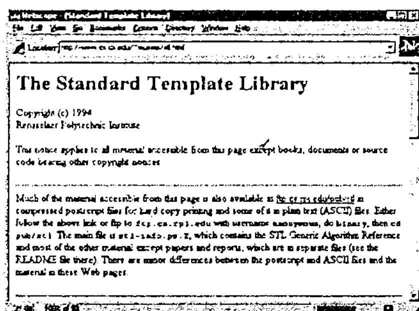
```
menu::clear_screen();
```

Ce trebuie să știți

În lecția de față ați învățat că precedarea unui membru de date al clasei cu cuvântul cheie *static* are ca efect partajarea acelui membru de către toate obiectele clasei respective. Dacă membrul de date este *public*, atunci programele pot accesa valoarea respectivului membru chiar și în cazul în care nu există nici un obiect al acelei clase. De asemenea, dacă precedați o metodă *publică* a unei clase cu cuvântul cheie *static*, programele vor putea utiliza acea funcție pentru operații care nu implică obiecte ale clasei. În lecția 28, „Despre moștenire", veți învăța să apelați la moștenire pentru a construi un obiect pe baza altor obiecte existente. Crearea de noi obiecte cu ajutorul moștenirii vă poate scuti de un efort considerabil de programare. Dar înainte de a trece la lecția 28, asigurați-vă că ați reținut următoarele aspecte importante:

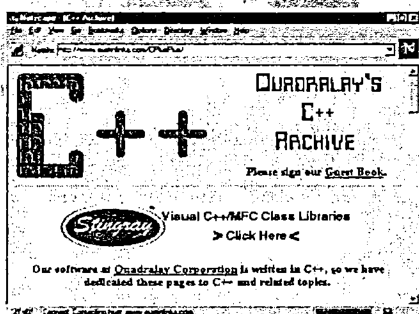
- ☑ Atunci când declarați un membru al unei clase ca *static*, acel membru este partajat de toate obiectele clasei respective.
- ☑ După declararea membrului unei clase ca *static*, programul trebuie să declare în exteriorul definiției de clasă o variabilă globală care corespunde membrului de date partajat.
- ☑ Un membru declarat ca *public* și *static* poate fi utilizat de program chiar și atunci când nu există nici un obiect al clasei respective. Pentru accesarea membrului, programul trebuie să utilizeze operatorul de rezoluție globală, ca de exemplu *nume_clasa::nume_membru*.
- ☑ O funcție membru declarată ca *publică* și *statică* poate fi apelată de program chiar și atunci când nu există nici un obiect al clasei respective. Pentru apelarea funcției, programul trebuie să utilizeze operatorul de rezoluție globală, ca de exemplu *menu::clear_screen()*.

Biblioteca de Șabloane Standard



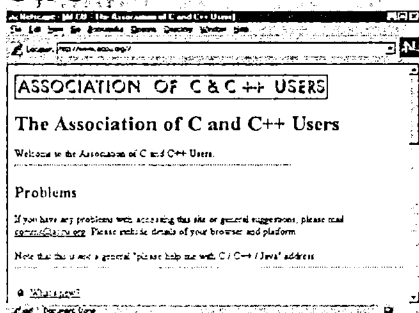
http://www.cs.rpi.edu/~musser/stl.html

Arhive C++



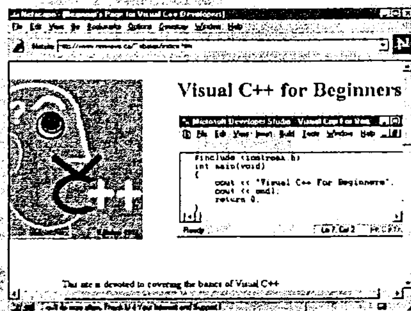
http://www.austinlinks.com/CPlusPlus/

Asociația Utilizatorilor C și C++



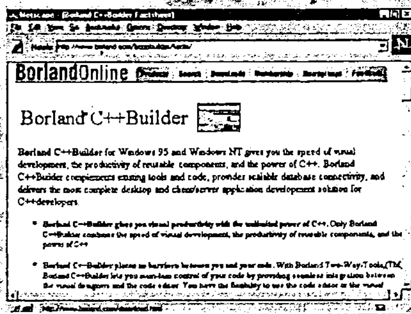
http://www.accu.org//

Pagina Începătorilor Visual C++



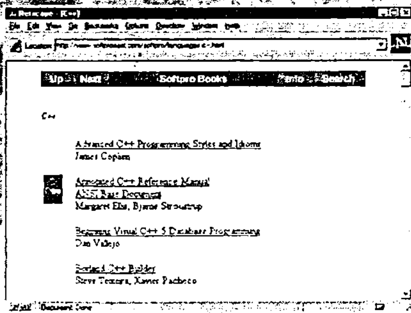
http://www.netwave.ca/~vbalas/index.html

Pagina Borland C++



http://www.borland.com/bcppbuilder/facts/

C++



http://www.softproeast.com/softpro/languages-c--.html

PARTEA a V-a

Despre moștenire și șabloane

Un avantaj major pe care îl aduce programarea orientată spre obiect este faptul că o clasă creată într-un program poate fi deseori folosită și în alte programe. Așa cum veți vedea în această parte a cărții, C++ permite nu numai refolosirea în programe a claselor, ci chiar construirea unei clase pe baza alteia. Atunci când creați o clasă pe baza unei alte clase, noua clasă va moșteni caracteristicile celei originale. În cadrul părții de față veți învăța să utilizați facilitățile de moștenire din C++ în scopul reducerii efortului de programare. În plus, veți afla cum se poate înlesni definirea tipurilor de date generice cu ajutorul șabloanelor din C++. Spre exemplu, prin intermediul unui șablon puteți defini o clasă generică *vector* pe care să o utilizați ulterior pentru a crea o variabilă ce reține valori de tip *int* și o altă variabilă ce reține valori de tip *float*. După parcurgerea acestei părți veți cunoaște o mulțime de lucruri despre conceptele programării orientate spre obiect. Lecțiile cuprinse în partea de față sunt următoarele:

Lecția 28

Despre moștenire

Programarea orientată spre obiect înlesnește refolosirea într-un program a unei clase pe care ați creat-o într-un alt program, economisindu-vă astfel timp și efort de programare. La definirea claselor, ar putea exista situații în care o clasă folosește multe sau chiar toate caracteristicile unei clase existente, adăugând apoi unul sau mai mulți membri de date sau funcții. În astfel de cazuri, C++ vă oferă posibilitatea de a crea noul obiect pe baza caracteristicilor obiectului existent. Cu alte cuvinte, noul obiect va *moșteni* membrii clasei existente (numită *clasă de bază*). Atunci când o nouă clasă este creată pe baza uneia deja existente, noua clasă este o *clasă derivată*. Lecția de față vă prezintă moștenirea în C++. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Atunci când utilizează moștenirea, programele folosesc o *clasă de bază* pentru a deriva o nouă clasă. Această clasă nouă (derivată) moștenește membrii clasei de bază.
- În scopul inițializării membrilor unei clase derivate, programul apelează funcțiile constructor ale clasei de bază și clasei derivate.
- Programele pot accesa în mod simplu membrii clasei de bază și ai clasei derivate prin intermediul operatorului punct.
- Pe lângă membrii *publici* (accesibili oricui) și cei *privați* (accesibili numai metodelor clasei), C++ permite declararea membrilor *protejați*, accesibili de către membrii clasei de bază și ai celei derivate.
- Pentru eliminarea conflictelor de nume dintre membrii clasei de bază și cei ai clasei derivate, programele pot utiliza operatorul de rezoluție globală (::), precedat de numele clasei de bază sau al celei derivate.

Moștenirea este un concept fundamental al programării orientate spre obiect. Experimentați o vreme cu programele prezentate în această lecție. După cum veți vedea, moștenirea este de fapt foarte ușor de implementat și vă poate scuti de multe ore de programare.

Un exemplu simplu de moștenire

Moștenirea permite clasei derivate să moștenească toate caracteristicile unei clase de bază existente. Să presupunem, de pildă, că avem următoarea clasă de bază *employee*:

```
class employee {
  public:
    employee(char *, char *, float);
    void show_employee(void);
  private:
    char name[64];
    char position[64];
    float salary;
};
```

Să presupunem, în continuare, că programul are nevoie de o clasă *manager*, care adaugă la clasa *employee* următorii membri:

```
float annual_bonus;
char company_car[64];
int stock_options;
```

În această situație există două soluții posibile. Pe de o parte, puteți crea în program o nouă clasă *manager* care va dubla o bună parte din membrii clasei *employee*. Pe de altă parte, puteți să derivați în program clasa *manager* din clasa de bază *employee*. Prin derivarea clasei *manager* din clasa *employee* deja existentă veți reduce efortul de programare și veți evita duplicarea de cod în program. Definiția noii clase începe cu cuvântul cheie *class*, urmat de numele *manager* și de semnul două puncte, după care se specifică numele *employee*, ca mai jos:

```
class manager  public employee {
    // Aici sunt definiti
    // membrii clasei
};
```

Clasa derivată

Clasa de bază

Cuvântul cheie *public* care precede numele clasei *employee* indică faptul că membrii *publici* ai clasei *employee* rămân *publici* și în cadrul clasei *manager*. Spre exemplu, instrucțiunile care urmează reprezintă derivarea clasei *manager*:

```
class manager  public employee {
  public:
    manager(char *, char *, char *, float, float, int);
    void show_manager(void);
```

```
   private:
     float annual_bonus;
     char company_car[64];
     int stock_options;
};
```

Atunci când derivați o clasă dintr-o clasă de bază, membrii *privați* din cadrul clasei de bază sunt accesibili clasei derivate numai prin intermediul funcțiilor de interfață oferite de clasa de bază. Din această cauză, o clasă derivată nu poate accesa direct un membru *privat* al clasei de bază prin utilizarea operatorului punct. Programul următor, *Mgr_Emp.CPP*, ilustrează utilizarea moștenirii în C++, construind clasa *manager* din clasa de bază *employee*:

```
#include <iostream.h>
#include <string.h>

class employee {
  public:
    employee(char *, char *, float);
    void show_employee(void);
  private:
    char name[64];
    char position[64];
    float salary;
};

employee::employee(char *name, char *position, float salary)
  {
    strcpy(employee::name, name);
    strcpy(employee::position, position);
    employee::salary = salary;
  }

void employee::show_employee(void)
  {
    cout << "Name: " << name << endl;
    cout << "Position: " << position << endl;
    cout << "Salary: $" << salary << endl;
  }
```

```
class manager : public employee {
  public:
  manager(char *, char *, char *, float, float, int);
  void show_manager(void);
  private:
  float annual_bonus;
  char company_car[64];
  int stock_options;
};

manager::manager(char *name, char *position,
  char *company_car,
    float salary, float bonus, int stock options) :
    employee(name, position, salary)
  {
    strcpy(manager::company_car, company_car);
    manager::annual_bonus = bonus;
    manager::stock_options = stock_options;
  }

void manager::show_manager(void)
  {
    show_employee();
    cout << "Company car: " << company_car << endl;
    cout << "Annual bonus: $" << annual_bonus << endl;
    cout << "Stock options: " << stock_options << endl;
  }

void main(void)
  {
    employee worker("John Doe", "Programmer", 35000);
    manager boss("Jane Doe", "Vice President", "Lexus",
       50000.0, 5000, 1000);

    worker.show_employee();
    boss.show_manager();
  }
```

După cum puteți vedea, programul definește clasa de bază *employee*, după care definește clasa derivată *manager*. Observați funcția constructor *manager*. La derivarea unei clase dintr-o clasă de bază, funcția constructor a clasei derivate trebuie să apeleze constructorul clasei de bază. Pentru a apela constructorul clasei de bază, plasați semnul două puncte imediat după funcția constructor a clasei derivate și specificați în continuare numele constructorului clasei de bază cu parametrii corespunzători, ca aici:

```
manager::manager(char *name, char *position,
  char *company_car,
    float salary, float bonus, int stock_options)
    employee(name, position, salary)
 {                                      Constructorul clasei de bază
    strcpy(manager::company_car, company_car)
    manager::annual_bonus = bonus;
    manager::stock_options = stock_options;
 }
```

Remarcați, de asemenea, cum funcția *show_manager* apelează funcția *show_employee*, aceasta din urmă fiind membră a clasei *employee*. Deoarece programul derivează clasa *manager* din clasa *employee*, clasa *manager* poate accesa membrii *publici* ai clasei *employee* ca și cum acești membri ar fi fost definiți în cadrul clasei *manager*.

Despre moștenire



Moștenirea reprezintă capacitatea unei clase derivate de a moșteni caracteristicile unei clase de bază existente. Sub o formă simplă, aceasta înseamnă că atunci când aveți o clasă ale cărei date sau funcții sunt necesare unei noi clase, este posibilă crearea acelei noi clase pe baza clasei existente (sau de bază). Noua clasă va moșteni astfel toți membrii (caracteristicile) clasei existente. Utilizarea moștenirii la construirea de noi clase vă economisește considerabil timpul și efortul de programare. Moștenirea este folosită frecvent în programarea orientată spre obiect, permițând programelor să construiască obiecte complexe pe baza unor obiecte mai mici și mai ușor de gestionat.

Un alt exemplu

Ca exemplu de moștenire, să presupunem că într-un program utilizați următoarea clasă de bază *book*:

```
class book {
  public:
  book(char *, char *, int);
  void show_book(void);
  private:
  char title[64];
  char author[64];
  int pages;
};
```

Să presupunem, mai departe, că programul necesită crearea unei clase *library_card* care va adăuga clasei *book* următorii membri:

```
char catalog[64];
int checked_out; // 1 daca este imprumutata, 0 in caz contrar
```

Programul poate recurge la moștenire pentru a deriva clasa *library_card* din clasa *book*, ca mai jos:

```
class library_card : public book {
  public:
  library_card(char *, char *, int, char *, int);
  void show_card(void);
private:
  char catalog[64];
  int checked_out;
};
```

Următorul program, *BookCard.CPP*, derivează clasa *library_card* din clasa *book*:

```
#include <iostream.h>
#include <string.h>

class book {
  public:
```

```
    book(char *, char *, int);
    void show_book(void);
  private:
    char title[64];
    char author[64];
    int pages;
  };

  book::book(char *title, char *author, int pages)
    {
      strcpy(book::title, title);
      strcpy(book::author, author);
      book::pages = pages;
    }

  void book::show_book(void)
    {
      cout << "Title: " << title << endl;
      cout << "Author: " << author << endl;
      cout << "Pages: " << pages << endl;
    }

  class library_card : public book {
    public:
      library_card(char *, char *, int, char *, int);
      void show_card(void);
    private:
      char catalog[64];
      int checked_out;
    };

  library_card::library_card(char *title, char *author,
    int pages, char *catalog, int checked_out)
    book(title, author, pages)
    {
    strcpy(library_card::catalog, catalog);
    library_card::checked_out = checked_out;
    }
```

```
void library_card::show_card(void)
  {
    show_book();
    cout << "Catalog: " << catalog << endl;
    if (checked_out)
      cout << "Status: Checked out" << endl;
    else
      cout << "Status: Available" << endl;
  }
void main(void)
  {
    library_card card("Rescued by C++", "Jamsa", 272,
      "101CPP", 1)
    card.show_card()
  }
```

Ca şi mai devreme, remarcaţi cum funcţia constructor *library_card* apelează constructorul clasei *book* pentru a iniţializa membrii acestei din urmă clase. De asemenea, remarcaţi utilizarea funcţiei membru *show_book* din clasa *book* în cadrul funcţiei *show_card*. Deoarece clasa *library_card* moşteneşte metodele clasei *book*, funcţia *show_card* poate apela o metodă fără a mai recurge la operatorul punct, ca şi cum funcţia *show_card* ar fi una din metodele definite în clasa *library_card*.

Despre membrii protejaţi

Atunci când întâlniţi diferite definiţii de clase de bază, membrii de clasă pe care îi găsiţi pot fi *publici*, *privaţi* sau *protejaţi*. Aşa cum ştiţi, o clasă derivată poate accesa membrii *publici* ai clasei de bază ca şi cum aceştia ar fi fost definiţi de program în cadrul clasei derivate. Pe de altă parte, o clasă derivată nu poate accesa direct membrii *privaţi* ai clasei de bază. În schimb, pentru a accesa aceşti membri, clasa derivată trebuie să apeleze la o funcţie de interfaţă a clasei de bază.

Un membru *protejat* al clasei de bază se situează undeva între un membru *privat* şi unul *public*. Dacă un membru este protejat, obiectele clasei derivate pot accesa acel membru ca şi cum ar fi fost *public*. Pentru restul programului, însă, membrii *protejaţi* apar ca *privaţi*. Singurul fel în care un program poate accesa membrii *protejaţi* este să apeleze la funcţiile de interfaţă. Următoarea definiţie a clasei *book* utilizează eticheta *protected* pentru a permite claselor derivate din clasa *book* să acceseze membrii *title*, *author* şi *pages* direct, prin intermediul operatorului punct.

```
class book {
  public:
    book(char *, char *, int);
  void show_book(void);
  protected:
    char title[64];
    char author[64];
    int pages;
};
```

În cazul în care credeți că este posibil să derivați mai târziu clase noi dintr-o clasă pe care o creați la momentul prezent, hotărâți dacă vreți să oferiți acelor clase posibilitatea de a accesa direct anumiți membri, iar apoi declarați membri corespunzători ca *protected*, în loc de *private*.

Membrii protejați

Așa cum ați aflat, membrii *protejați* ai unei clase previn accesarea directă de către program a membrilor respectivi. Pentru a accesa un membru *protejat* al clasei, programul trebuie să apeleze, în schimb, la o funcție de interfață care controlează accesul la acel membru. Atunci când utilizați moștenirea într-un program, ați putea considera că accesarea de către clasele derivate a unor membri din clasa de bază prin intermediul operatorului punct duce la simplificarea programării. În astfel de cazuri, puteți recurge la declararea în program a membrilor *protejați*. O clasă derivată poate accesa direct un membru *protejat* prin intermediul operatorului punct. Restul programului, însă, va putea accesa membrii *protejați* ai clasei numai prin intermediul funcțiilor de interfață oferite de către aceasta. Membrii de clasă *protejați* se află undeva între membrii de clasă *publici* (care pot fi accesați oriunde în program) și membrii de clasă *privați*, care pot fi accesați direct numai de către clasa care-i conține.

Specificarea numelor de membri

Atunci când derivați o clasă dintr-o altă clasă, este posibil ca numele unui membru din clasa derivată să fie același cu numele unui membru din clasa de bază. În cazul în care apare un astfel de conflict, C++ utilizează întotdeauna în funcțiile clasei derivate membrii acesteia. De exemplu, să presupunem că clasele *book* și *library_card* conțin ambele un membru *price*. În cazul clasei *book*, membrul *price* corespunde prețului de vânzare al unei cărți, ca de pildă $29.95. În cazul clasei *library_card*, valoarea *price* ar putea include o reducere, de exemplu $24.50. În cazul în care nu se precizează altceva (prin intermediul operatorului de rezoluție gloablă), funcțiile clasei *library_card* vor utiliza membrii clasei derivate (*library_card*). Dacă o funcție a clasei *library_card* necesită accesarea membrului *price* din clasa de bază (*book*), atunci respectiva funcție poate utiliza numele clasei *book* și

operatorul de rezoluție, cum ar fi *book::price*. Să presupunem, de exemplu, că funcția *show_card* trebuie să afișeze ambele prețuri. Funcția va conține atunci următoarele instrucțiuni:

```
cout << "Library price: $" << price << endl;
cout << "Retail price: $" << book::price << endl;
```

Dacă executați programul, instrucțiunile de mai sus vor afișa următoarele:

```
Library price: $24.50
Retail price: $29.95
```

Ce trebuie să știți

În lecția de față ați văzut cum facilitățile de moștenire din C++ vă permit construirea (derivarea) unei clase noi pe baza unei clase deja existente. O astfel de creare a unei clase pe baza alteia duce la reducerea efortului de programare, ceea ce duce mai departe la o economie de timp. În lecția 29, „Moștenirea multiplă", veți vedea că C++ vă permite să derivați o clasă din două sau mai multe clase de bază. Utilizarea mai multor clase de bază pentru derivarea unei clase poartă numele de *moștenire multiplă*. Dar înainte de a trece la lecția 29, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Moștenirea reprezintă capacitatea de a deriva o clasă nouă dintr-o clasă de bază existentă.

☑ Clasa derivată este clasa cea nouă, iar clasa de bază este clasa originală.

☑ Atunci când derivați o clasă dintr-o alta (clasa de bază), clasa derivată moștenește membrii clasei de bază.

☑ Derivarea unei clase dintr-o clasă de bază se face prin definirea clasei cu cuvântul cheie *class*, urmat de numele clasei, semnul două puncte și clasa de bază, ca în *class dalmatian : caine*.

☑ Atunci când derivați o clasă dintr-o clasă de bază, clasa derivată poate accesa membrii *publici* ai clasei de bază ca și cum respectivii membri ar fi fost definiți de program în cadrul clasei derivate. Pentru accesarea datelor *private* din clasa de bază, clasa derivată trebuie să apeleze la funcțiile de interfață oferite de clasa de bază.

☑ În cadrul funcției constructor a clasei derivate, programul trebuie să apeleze constructorul clasei de bază prin plasarea imediat după antetul funcției constructor din clasa derivată a semnului două puncte, urmat de numele funcției și parametrii corespunzători.

☑ Pentru a oferi claselor derivate un acces direct la anumiţi membri ai clasei de bază, protejând totodată membrii respectivi de restul programului, C++ permite definirea de membrii de clasă *protejaţi*. O clasă derivată poate să acceseze membrii protejaţi ai clasei de bază ca şi când aceştia ar fi *publici*. Pentru restul programului, însă, membrii respectivi apar ca *privaţi*.

☑ Dacă o clasă derivată şi o clasă de bază conţin membri ce au acelaşi nume, C++ va utiliza în cadrul funcţiilor din clasa derivată membrii aparţinând acestei clase. În cazul în care o astfel de funcţie trebuie să acceseze un membru al clasei de bază, va trebui să utilizaţi operatorul de rezoluţie globală, sub forma *clasa_de_baza::membru*.

Lecția 29

Moștenirea multiplă

În lecția 28, „Despre moștenire", ați văzut cum puteți construi o clasă pe baza unei alte clase prin moștenirea caracteristicilor acesteia din urmă. În fapt, C++ vă permite să derivați o clasă din mai multe clase de bază. Atunci când o clasă moștenește caracteristicile mai multor clase, acea nouă clasă se obține printr-o *moștenire multiplă.* După cum veți vedea în lecția de față, moștenirea multiplă este perfect posibilă în C++. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Derivarea unei clase din mai multe clase de bază reprezintă o *moștenire multiplă.*
- Prin intermediul moștenirii multiple, o clasă derivată preia atributele a două sau mai multe clase de bază.
- Atunci când derivați o clasă prin intermediul moștenirii multiple, constructorul clasei derivate trebuie să apeleze funcțiile constructor ale fiecăreia din clasele de bază.
- Derivarea unei clase dintr-o clasă derivată duce la crearea unui lanț de derivare.

Moștenirea multiplă este o facilitate foarte puternică a programării orientate spre obiect. Experimentați cu programele prezentate în această lecție. Precum veți vedea, construirea unei clase pe baza unei clase existente poate reduce substanțial efortul de programare.

Inspectarea unui exemplu simplu

Ca exemplu de moștenire multiplă, să presupunem că avem următoarea clasă *computer_screen*:

```
class computer_screen {
  public:
    computer_screen(char *, long, int, int);
    void show_screen(void)
  private:
    char type[32];
    long colors;
    int x_resolution;
    int y_resolution;
};
```

De asemenea, să presupunem că avem următoarea clasă *mother_board*:

```
class mother_board {
  public:
    mother_board(int, int, int);
    void show_mother_board(void);
  private:
    int processor;
    int speed;
    int RAM;
};
```

Cu ajutorul acestor două clase putem deriva o clasă *computer* ilustrată aici:

```
class computer : public computer_screen,
  public mother_board {
  public:
    computer(char *, int, float, char *, long,
      int, int, int, int, int);
    void show_computer(void);
  private:
    char name[64];
    int hard_disk;
    float floppy;
};
```

După cum puteți vedea, această clasă precizează clasele sale de bază imediat după semnul două puncte care urmează numelui său de clasă, *computer*:

```
class computer : public computer_screen, public mother_board
```

Clase de bază

Programul următor, *Computer.CPP*, derivează clasa *computer* utilizând clasele de bază *computer_screen* și *mother_board*:

```
#include <iostream.h>
#include <string.h>
```

```
class computer_screen {
  public:
    computer_screen(char *, long, int, int);
    void show_screen(void);
  private:
    char type[32];
    long colors;
    int x_resolution;
    int y_resolution;
}

computer_screen::computer_screen(char *type,
  long colors, int x_res, int y_res)
  {
    strcpy(computer_screen::type, type)
    computer_screen::colors = colors;
    computer_screen::x_resolution = x_res;
    computer_screen::y_resolution = y_res;
  }

void computer_screen::show_screen(void)
  {
    cout << "Screen type: " << type << endl;
    cout << "Colors:   << colors << endl;
    cout << "Resolution:   << x_resolution << " by   <<
      y_resolution << endl;
  }

class mother_board {
  public:
    mother_board(int, int, int);
    void show_mother_board(void);
  private:
    int processor;
    int speed;
    int RAM;
};
```

```
mother_board::mother_board(int processor, int speed, int RAM)
  {
    mother_board::processor = processor;
    mother_board::speed = speed;
    mother_board::RAM = RAM;
  }

void mother_board::show_mother_board(void)
  {
    cout << "Processor: " << processor << endl;
    cout << "Speed: " << speed << "Mhz" << endl;
    cout << "RAM: " << RAM << " Mb" << endl;
  }

class computer : public computer_screen,
    public mother_board {
  public:
    computer(char *, int, float, char *, long,
      int, int, int, int, int);
    void show_computer(void);
  private:
    char name[64];
    int hard_disk;
    float floppy;
};

computer::computer(char *name, int hard_disk,
  float floppy, char *screen, long colors,
  int x_res, int y_res, int processor, int speed,
  int RAM) : computer_screen(screen, colors, x_res,
  y_res), mother_board(processor, speed, RAM)
  {
    strcpy(computer::name, name);
    computer::hard_disk = hard_disk;
    computer::floppy = floppy;
  }
```

```
void computer::show_computer(void)
  {
    cout << "Type: " << name << endl;
    cout << "Hard disk: " << hard_disk << "Mb" << endl;
    cout << "Floppy disk: " << floppy << "Mb" << endl;
    show_mother_board();
    show_screen();
  }

void main(void)
  {
    computer my_pc("Compaq", 212, 1.44, "SVGA",
      16000000, 640, 480, 486, 66, 8);

    my_pc.show_computer();
  }
```

Dacă priviți funcția constructor a clasei *computer*, puteți vedea că aceasta apelează funcțiile constructor ale claselor *mother_board* și *computer_screen*, ca aici:

```
computer::computer(char *name, int hard_disk,
  float floppy, char *screen, long colors,
  int x_res, int y_res, int processor, int speed,
  int RAM) : computer_screen(screen, colors, x_res,
  y_res), mother_board(processor, speed, RAM)
```

Crearea unei ierarhii de clase

Atunci când recurgeți la moștenirea din C++ pentru a deriva o clasă dintr-o alta, pot exista situații în care să derivați o clasă dintr-o altă clasă, care este, la rândul său, derivată în program dintr-o clasă de bază. Să presupunem, de exemplu, că vreți să folosiți clasa *computer* ca și clasă de bază pentru a deriva clasa *work_station* prezentată aici:

```
class work_station : public computer {
  public:
    work_station(char *operating_system, char *name,
      int hard_disk, float floppy, char *screen,
      long colors, int x_res, int y_res,
      int processor, int speed, int RAM);
```

```
    void show_work_station(void);
  private:
    char operating_system[64];
};
```

Funcția constructor a clasei *work_station* apelează doar constructorul clasei *computer*, iar acesta apelează, la rândul său, constructorii claselor *computer_screen* și *mother_board*:

```
work_station(char *operating_system,
  char *name, int hard_disk, float floppy,
  char *screen, long colors, int x_res, int y_res,
  int processor, int speed, int RAM) :
  computer (name, hard_disk, floppy, screen,
    colors, x_res, y_res, processor, speed, RAM)
  {
    strcpy(work_station::operating_system, operating_system)
  }
```

În exemplul de mai sus, clasa *computer* este folosită ca și clasă de bază. După cum știți, însă, clasa *computer* a fost derivată din clasele *computer_screen* și *mother_board*. Prin urmare, clasa *work_station* moștenește caracteristicile tuturor celor trei clase. Așa cum o ilustrează și figura 29, derivarea claselor formează o ierarhie de clase.

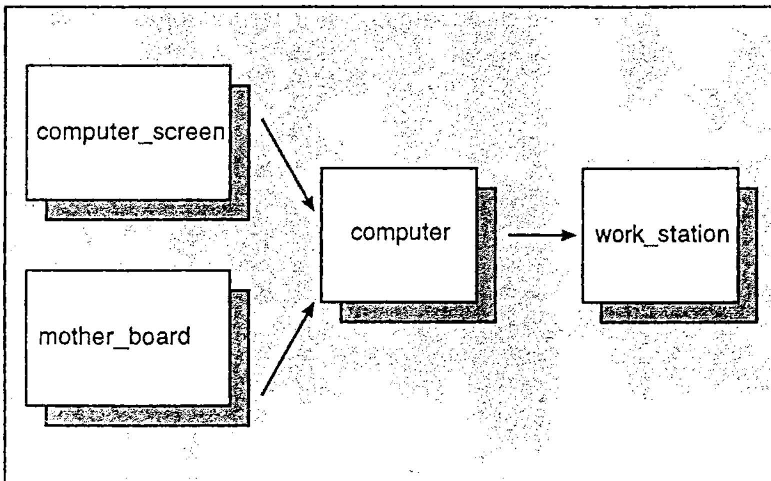


Figura 29 Formarea unei ierarhii de clase.

Pe măsură ce programele dumneavoastră vor apela tot mai mult la moștenire, ierarhiile de clase și, implicit, numărul membrilor moșteniți de o clasă pot crește considerabil.

Ce trebuie să știți

Moștenirea multiplă reprezintă capacitatea de a deriva o clasă din două sau mai multe clase de bază. La utilizarea moștenirii multiple, clasa derivată preia (moștenește) caracteristicile (membrii) claselor de bază existente. Suportul oferit de C++ pentru moștenirea multiplă aduce facilități deosebite în ceea privește programarea orientată spre obiect. În lecția 30, „Membri privați și prieteni", veți vedea cum puteți permite accesarea membrilor *privați* ai unei clase de către alte clase sau de funcții ale altor clase care sunt definite ca *prieteni*. Prin intermediul unor astfel de prieteni puteți să oferiți anumitor funcții acces direct la membrii unei clase, protejând în continuare acei membri față de restul programului. Dar înainte de a trece la lecția 30, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Moștenirea multiplă denotă capacitatea unei clase derivate de a moșteni caracteristicile a două sau mai multe clase de bază.

☑ La derivarea unei clase din două sau mai multe clase de bază trebuie specificate numele claselor de bază, separate de virgulă, plasate după numele noii clase și semnul două puncte, ca de pildă *class crocobaur : public crocodil, public balaur.*

☑ La definirea funcției constructor a clasei derivate trebuie să apelați funcțiile constructor ale fiecărei clase de bază, transmițând acestor funcții parametrii adecvați.

☑ Atunci când derivați o clasă, este posibil, uneori, ca clasa de bază pe care o utilizați să fie la rândul său derivată din alte clase de bază. În asemenea situații se formează o ierarhie de clase. Atunci când programul apelează constructorul unei clase derivate, C++ apelează de asemenea funcțiile constructor ale claselor moștenite (în ordinea corespunzătoare).

Lecția 30
Membri privați și prieteni

După cum ați aflat, programele pot accesa membrii *privați* ai unei clase exclusiv prin intermediul funcțiilor membre ale acelei clase. Prin utilizarea cât mai frecventă a membrilor *privați* reduceți șansele ca un program să folosească eronat valoarea unui membru, deoarece programul nu va putea accesa acel membru decât cu ajutorul unei funcții de interfață (care poate controla accesul la membrul respectiv). În funcție de modul de folosire al obiectelor dintr-un program, însă, pot exista cazuri în care performanțele programului sporesc semnificativ atunci când o clasă are posibilitatea de a accesa direct membrii *privați* ai unei alte clase. În acest fel, programele evită efortul de calcul (timpul de execuție) implicat de apelul funcțiilor de interfață. Pentru asemenea situații, C++ vă oferă posibilitatea de a defini o clasă ca fiind *prietena* unei alte clase, permițând apoi clasei *prietene* să acceseze membrii *privați* ai celeilalte clase. Lecția de față studiază modul în care se specifică într-un program această relație de prietenie dintre două clase. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Cu ajutorul cuvântului cheie *friend*, o clasă poate informa C++ care sunt prietenii săi – cu alte cuvinte, ce alte clase pot accesa direct membrii săi *privați*.
- Membrii *privați* ai unei clase protejează datele acesteia; acesta este motivul pentru care este bine să limitați folosirea claselor *prietene* la acele cazuri în care este cu adevărat necesar ca o clasă să acceseze direct membrii *privați* ai unei alte clase.
- C++ vă permite restrângerea accesului unei clase *prietene* la o anumită mulțime de funcții.

Membrii *privați* permit protejarea claselor și reduc probabilitatea de apariție a erorilor. Din această cauză este recomandat să recurgeți cât mai puțin la clase *prietene*. De fiecare dată când un program poate modifica direct membrii unei clase, probabilitatea de apariție a erorilor crește corespunzător.

Definirea unei clase prietene

C++ permite unei clase să acceseze membrii *privați* ai clasei căreia îi este prietenă. Pentru a informa C++ că o clasă este *prietenă* a unei alte clase, este suficient să precizați în definiția clasei din urmă cuvântul cheie *friend* și numele clasei prietene corespunzătoare. De exemplu, clasa *book* de mai jos declară clasa *librarian* ca fiind *prietenă*. În consecință, obiectele clasei *librarian* pot accesa direct membrii *privați* ai clasei *book* prin intermediul operatorului punct:

```
class book {
  public:
    book(char *, char *, char *);
    void show_book(void);
    friend librarian;        // Clasa prietena
  private:
    char title[64];
    char author[64];
    char catalog[64];
}
```

După cum puteți vedea, specificarea unui *prieten* nu presupune decât o singură instrucțiune în definiția unei clase. Spre exemplu, programul următor, *ViewBook.CPP*, utilizează *librarian* ca *prietenă* a clasei *book*. Astfel, funcțiile clasei *librarian* pot accesa direct membrii *privați* ai clasei *book*. În acest exemplu, programul folosește funcția *change_catalog* a clasei *librarian* pentru a modifica numărul de catalog al unei anumite cărți:

```
#include <iostream.h>
#include <string.h>

class librarian;

class book {
  public:
    book(char *, char *, char *);
    void show_book(void)
    friend librarian;        // Clasa prietena
  private:
    char title[64];
    char author[64];
    char catalog[64];
};

book::book(char *title, char *author, char *catalog)
  {
    strcpy(book::title, title);
    strcpy(book::author, author);
    strcpy(book::catalog, catalog);
  }
```

```
void book::show_book(void)
  {
    cout << "Title:    << title << endl;
    cout << "Author:    << author << endl;
    cout << "Catalog: " << catalog << endl;
  }

class librarian {
  public:
    void change_catalog(book *, char *);
    char *get_catalog(book);
};

void librarian::change_catalog(book *this_book,
  char *new_catalog)
  {
    strcpy(this_book->catalog, new_catalog);
  }

char *librarian::get_catalog(book this_book)
  {
    static char catalog[64];
    strcpy(catalog, this_book.catalog);
    return(catalog);
  }

void main(void)
  {
    book programming("Rescued By C++, Third Edition",
      "Jamsa", "P101");
    librarian library;

    programming.show_book();
    library.change_catalog(&programming, "EASY C++ 101");
    programming.show_book();
  }
```

Precum vedeți, programul *ViewBook.CPP* transmite obiectul *book* către funcția *change_catalog* a clasei *librarian* prin adresă. Deoarece funcția modifică un membru al clasei, programul trebuie să transmită parametrul prin adresă și să folosească apoi un pointer

pentru a accesa membrul respectiv. Experimentați cu acest program, înlăturând declarația *friend* din definiția clasei *book*. Deoarece clasa *librarian* nu mai are acces la membrii *privați* ai clasei *book*, compilatorul de C++ va genera erori de sintaxă pentru fiecare caz în care programul încearcă să apeleze datele *private* din clasa *book*.

Despre clasele prietene



În mod normal, singurul fel în care programele pot accesa datele *private* ale unei clase este să apeleze la funcțiile de interfață ale acelei clase. În funcție de modul de folosire al obiectelor din program, însă, ar putea exista situații în care să fie mai convenabil (sau mai eficient din perspectiva vitezei de lucru) ca unei clase să i se acorde accesul la membrii *privați* ai alteia. În acest scop, va trebui să informați compilatorul C++ că acea clasă este o *prietenă*. Compilatorul, la rândul său, va permite clasei *prietene* să acceseze direct membrii *privați*. Pentru a defini o clasă ca *prietenă*, plasați între membrii *publici* din definiția unei clase cuvântul cheie *friend* și numele clasei corespunzătoare, așa cum este ilustrat aici:

```
class tanta {
  public:
    friend costel;
    // Alti membri
  private:
    // Membri privati
};
```

Diferența dintre implicațiile claselor prietene și cele ale membrilor protejați



În lecția 28, „Despre moștenire", ați aflat că C++ acceptă membri *protejați*, ceea ce permite claselor derivate să acceseze membrii *privați* ai unei clase de bază direct, prin intermediul operatorului punct. Rețineți că singurele clase care pot accesa membrii *protejați* ai unei clase sunt clasele pe care programul le derivă din clasa de bază – cu alte cuvinte, clasele care moștenesc membrii clasei de bază. Clasele *prietene* din C++ sunt, de regulă, clase independente (adică nici o clasă nu moștenește membrii celeilalte). Singura modalitate în care o astfel de clasă independentă poate accesa membrii *privați* ai unei alte clase este ca aceasta din urmă să informeze compilatorul că respectiva clasă independentă este *prietenă*.

Restricționarea accesului unei clase prietene

Așa cum ați văzut, declararea unei clase ca fiind *prietena* altei clase oferă primei clase acces la datele private ale celei din urmă. Numai că odată cu lărgirea accesului la datele *private* ale claselor cresc și șansele apariției de erori în programe. Atunci, în cazul în care numai câteva funcții dintr-o clasă *prietenă* au nevoie de acces la datele *private* ale unei alte clase, C++ vă permite să restrângeți în program accesul la membrii *privați* doar pentru anumite funcții *prietene* dintr-o clasă. Să presupunem, de exemplu, că clasa *librarian* prezentată în programul anterior conține multe funcții diferite. De asemenea, să presupunem că funcțiile *change_catalog* și *get_catalog* sunt singurele care au nevoie de acces la datele *private* ale clasei *book*. În definiția clasei *book* este posibilă restricționarea accesului la membrii privați pentru doar aceste două funcții, ca mai jos:

```
class book {
  public:
    book(char *, char *, char *);
    void show_book(void);
    friend char *librarian::get_catalog(book)
    friend void librarian::change_catalog(book *, char *)
  private:
    char title[64];
    char author[64];
    char catalog[64];
};
```

Așa cum puteți vedea, instrucțiunile *friend* conțin prototipurile complete pentru fiecare din funcțiile *prietene* care vor putea accesa direct membrii *privați*.

Despre funcțiile prietene



În cazul în care un program folosește relații de prietenie pentru a accesa datele *private* ale unei clase, puteți limita numărul funcțiilor membru ale clasei *prietene* care pot accesa date *private* prin utilizarea de *funcții prietene*. Declararea unei funcții *prietene* se face prin precizarea cuvântului cheie *friend*, urmat de prototipul complet al funcției care va accesa date, așa cum este ilustrat aici:

```
public:
    friend nume_clasa::nume_functie(tipurile parametrilor);
```

Numai funcțiile membru pe care la declarați ca *prietene* vor putea accesa membrii *privați* ai clasei în mod direct, cu ajutorul operatorului punct.

Atunci când într-un program există o clasă ce conține referințe la o altă clasă, o ordine necorespunzătoare a definițiilor va duce la producerea de erori de sintaxă. În cazul nostru, definiția clasei *book* folosește prototipurile funcțiilor definite în clasa *librarian*. Din această cauză, definiția clasei *librarian* trebuie să preceadă definiția clasei *book*. Dacă priviți, însă, clasa *librarian*, veți vedea că aceasta conține referințe la clasa *book*, ca aici:

```
class librarian {
  public:
    void change_catalog(book *, char *);
    char *get_catalog(book);
};
```

Deoarece nu este posibil ca definițiile claselor *book* și *librarian* să se preceadă una pe cealaltă, C++ vă permite să specificați o definiție de clasă pe o singură linie care informează compilatorul că *book* este o clasă pe care programul o va defini mai târziu:

```
class book;
```

Programul următor, *LimitFri.CPP*, utilizează funcții *prietene* pentru a limita accesul clasei *librarian* la datele *private* ale clasei *book*. Remarcați ordinea definițiilor de clase:

```
#include <iostream.h>
#include <string.h>

class book;

class librarian {
  public:
    void change_catalog(book *, char *);
    char *get_catalog(book);
};

class book {
  public:
    book(char *, char *, char *);
    void show_book(void);
    friend char *librarian::get_catalog(book);
    friend void librarian::change_catalog(book *, char *);
  private:
    char title[64];
    char author[64];
```

```
    char catalog[64];
};

book::book(char *title, char *author, char *catalog)
  {
    strcpy(book::title, title);
    strcpy(book::author, author);
    strcpy(book::catalog, catalog);
  }

void book::show_book(void)
  {
    cout << "Title: " << title << endl;
    cout << "Author:   << author << endl;
    cout << "Catalog: " << catalog << endl;
  }

void librarian::change_catalog(book *this book,
  char *new_catalog)
  {
    strcpy(this_book->catalog, new_catalog);
  }

char *librarian::get_catalog(book this_book)
  {
    static char catalog[64];

    strcpy(catalog, this_book.catalog);
    return(catalog);
  }

void main(void)
  {
    book programming("Rescued By C++, Third Edition",
      "Jamsa", "P101");
    librarian library;

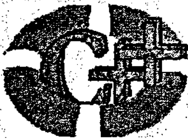
    programming.show_book();

    library.change_catalog(&programming, "EASY C+ 101");
```

```
    programming.show_book();
}
```

Așa cum se poate vedea, programul *LimitFri.CPP* conține mai întâi o linie care informează compilatorul că definiția clasei *book* va fi efectuată ulterior. Deoarece această declarație aduce la cunoștința compilatorului existența unei clase *book*, definiția clasei *librarian* poate conține referiri la o clasă *book* pe care programul urmează să o definească mai târziu.

Specificarea unui identificator de clasă



Un identificator este un nume, precum o variabilă sau un tip de clasă. Atunci când programele utilizează clase *prietene*, pot exista situații în care definiția unei clase face referire la o altă clasă (prin numele sau identificatorul acesteia) despre care compilatorul C++ nu știe nimic. În astfel de cazuri, compilatorul de C++ generează mesaje de eroare de sintaxă. Pentru a elimina astfel de erori de genul „cine este definit mai întâi", C++ vă permite să precizați o declarație pe o singură linie, ilustrată aici, plasată în partea superioară a codului și care specifică un identificator de clasă:

```
class nume_clasa;
```

Declarația informează compilatorul de C++ că programul va defini mai târziu clasa precizată, iar pentru moment este în regulă ca programul să facă referire la numele acestei clase.

Ce trebuie să știți

În lecția de față ați învățat să utilizați clasele *prietene* pentru a permite unei clase să acceseze direct, prin intermediul operatorului punct, membrii *privați* ai unei alte clase. În lecția 31, „Utilizarea șabloanelor de funcții", veți învăța să folosiți șabloanele de funcții din C++ pentru a simplifica definirea funcțiilor similare. Dar înainte de a trece la lecția 31, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Cu ajutorul relației de prietenie din C++, programele pot permite unei clase să acceseze membrii *privați* ai altei clase în mod direct, prin specificarea operatorului punct.

- ☑ Pentru a declara o clasă ca *prietenă* a unei alte clase, specificați în definiția clasei din urmă cuvântul cheie *friend*, urmat de numele primei clase.

☑ După declararea unei clase ca *prietenă* a unei alte clase, toate funcţiile membru ale clasei *prietene* pot accesa membrii *privaţi* ai celeilalte clase.

☑ Pentru a limita numărul metodelor *prietene* care pot accesa datele *private* ale unei clase, C++ vă permite declararea de funcţii *prietene*. Declararea unei funcţii *prietene* se face prin specificarea cuvântului cheie *friend*, urmat de prototipul funcţiei care necesită accesul la membrii *privaţi* ai clasei.

☑ La declararea funcţiilor prietene, este posibil să întâlniţi erori de sintaxă în cazul în care ordinea definiţiilor de clase din program nu este adecvată. Dacă trebuie să informaţi C++ că un anumit identificator reprezintă o clasă pe care programul o va defini mai târziu, atunci puteţi preciza o declaraţie pe o singură linie de forma *class nume_clasa;*.

Lecția 31

Utilizarea șabloanelor de funcții

Atunci când scrieți funcții în cadrul programelor, ar putea exista situații în care două funcții efectuează operații similare, dar lucrează cu tipuri diferite de date (de exemplu, o funcție are parametri de tip *int*, iar cealaltă de tip *float*). Așa cum ați văzut în lecția 14, „Supradefinirea funcțiilor“, prin supradefinirea funcțiilor puteți să folosiți același nume de funcție pentru a îndeplini operații care folosesc parametri de tipuri diferite. Când funcțiile întorc valori diferite, însă, trebuie să definiți funcții cu nume unice. Să presupunem, de pildă, că aveți o funcție numită *max* care întoarce maximul dintre două valori întregi. Dacă mai târziu doriți o funcție similară care să întoarcă maximul dintre două valori în virgulă mobilă, va trebui să definiți o nouă funcție, precum *fmax*. În lecția de față veți învăța să apelați la *șabloanele* din C++ pentru a crea rapid funcții care întorc valori de tipuri diferite. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- *Șabloanele* definesc o mulțime de instrucțiuni pe care programul le poate apela mai târziu pentru a crea noi funcții.
- Programele folosesc deseori *șabloane de funcții* pentru a defini rapid două sau mai multe funcții care conțin instrucțiuni similare, dar care au tipuri diferite ale parametrilor sau ale valorilor întoarse.
- Șabloanele de funcții au nume specifice care sunt identice cu numele funcțiilor pe care vreți să le utilizați în program.
- După definirea unui șablon de funcție, programul poate crea ulterior o funcție prin utilizarea șablonului pentru a specifica un prototip ce include numele acestuia, valoarea întoarsă de funcție și tipurile parametrilor funcției.
- În timpul compilării, compilatorul de C++ va crea în program funcțiile corespunzătoare pe baza tipurilor specificate în cadrul prototipurilor care folosesc numele unui șablon.

Șabloanele de funcții folosesc o sintaxă unică ce v-ar putea intimida. Dar după utilizarea a unul sau două șabloane veți descoperi că acestea sunt de fapt foarte ușor de folosit.

Notă: *Compilatorul* ***Turbo C++ Lite*** *oferit pe CD-ROM-ul ce însoțește această carte nu oferă suport pentru utilizarea șabloanelor din C++.*

Crearea unui șablon de funcție simplu

Un *șablon de funcție* definește o funcție fără tipuri (generică) pe care programele o pot utiliza mai târziu pentru a crea funcții care folosesc tipuri specificate prin program. De exemplu, instrucțiunile următoare definesc un șablon pentru o funcție numită *max*, care întoarce cea mai mare dintre două valori:

```
template<class T> T max(T a, T b)
  {
    if (a > b)
      return(a);
    else
      return(b);
  }
```

În acest exemplu, litera *T* reprezintă tipul generic al şablonului. După definirea în program a acestui şablon, declaraţi prototipuri de funcţii corespunzătoare fiecărui tip necesar. În cazul şablonului *max*, prototipurile următoare creează funcţii de tip *float* şi, respectiv, *int*.

```
float max(float, float);
int max(int, int);
```

Atunci când compilatorul de C++ întâlneşte în program aceste prototipuri, el va construi funcţiile specificate, înlocuind tipul *T* din şablon cu tipul corespunzător. În cazul implementării cu *float* a funcţiei *max*, înlocuirile efectuate de compilator se prezintă astfel:

```
float max(float, float);

      template<class T> T max(T a, T b)
      {
        if (a > b)
          return(a);
        else
          return(b);
      }

          float max(float a, float b)
          {
            if (a > b)
              return(a);
          else
              return(b);
          }
```

Programul următor, *Max_Temp.CPP*, utilizează șablonul *max* pentru a crea funcții de tip *int* și *float*:

```
#include <iostream.h>

template<class T> T max(T a, T b)
  {
    if (a > b)
      return(a);
    else
      return(b);
  }

float max(float, float);
int max(int, int);

void main(void)
  {
    cout << "The maximum of 100 and 200 is " <<
      max(100, 200) << endl;
    cout << "The maximum of 5.4321 and 1.2345 is " <<
      max(5.4321, 1.2345) << endl;
  }
```

La compilare, compilatorul de C++ creează automat instrucțiunile care alcătuiesc funcția corespunzătoare tipului *int* și pe cele care alcătuiesc funcția corespunzătoare tipului *float*. Deoarece compilatorul de C++ este cel care gestionează instrucțiunile corespunzătoare funcțiilor pe care le creați pe baza șabloanelor, vă este permisă utilizarea aceluiași nume pentru funcții care întorc tipuri diferite, ceea ce nu ar fi fost posibil prin intermediul supradefinirii funcțiilor (despre care discutam în lecția 14).

Șabloane care folosesc mai multe tipuri

Definiția de mai devreme pentru șablonul de funcție *max* folosea un singur tip generic, *T*. De multe ori, însă, un șablon de funcție poate necesita mai multe tipuri. Spre exemplu, instrucțiunile care urmează creează un șablon pentru funcția *show_array*, care afișează elementele unui vector. Șablonul folosește tipul *T* pentru a defini tipul vectorului și tipul *T1* pentru a specifica tipul parametrului count:

Utilizarea şabloanelor de funcţii



Pe măsură ce programele pe care le creaţi sporesc în complexitate, vor apărea multe situaţii în care veţi avea nevoie de funcţii similare care să efectueze aceleaşi operaţii, dar asupra unor tipuri de date diferite. Şabloanele de funcţii permit programelor să definească o funcţie generică, sau fără tipuri. Atunci când trebuie să folosească o funcţie pentru un anumit tip, precum *int* sau *float*, programul specifică un prototip de funcţie care conţine numele şablonului de funcţie şi care precizează tipul valorii întoarse de funcţie şi al fiecăruia dintre parametrii acesteia. Compilatorul de C++ va crea în timpul compilării funcţiile corespunzătoare. Prin folosirea şabloanelor de funcţii reduceţi numărul funcţiilor pe care trebuie să le scrieţi, iar programele pot folosi un acelaşi nume pentru funcţiile care îndeplinesc o anumită operaţie, indiferent de tipul valorii întoarse sau al parametrilor acelei funcţii.

```
template<class T, class T1> void show_array(T *array,
  T1 count)
 {
   T1 index;

   for (index = 0; index < count; index++)
     cout << array[index] << ' ';

   cout << endl;
 }
```

Ca şi mai devreme, programul trebuie să specifice prototipurile funcţiilor care corespund fiecărei perechi de tipuri:

```
void show_array(int *, int);
void show_array(float *, unsigned);
```

Programul următor, *Show_Tem.CPP*, utilizează şablonul *show_array* pentru a crea funcţii care afişează vectori de tip *int* şi, respectiv, *float*:

```
#include <iostream.h>

template<class T, class T1> void show_array(T *array,
  T1 count)
```

```
  {
    T1 index;

    for (index = 0; index < count; index++)
      cout << array[index] << ' ';

    cout << endl;
  }

void show_array(int *, int);
void show_array(float *, unsigned);

void main(void)
  {
    int pages[] = { 100, 200, 300, 400, 500 };
    float prices[] = { 10.05, 20.10, 30.15 };

    show_array(pages, 5);
    show_array(prices, 3);
}
```

Șabloane și tipuri multiple



Pe măsură ce șabloanele de funcții create vor crește în complexitate, acestea ar putea accepta mai multe tipuri de date. De exemplu, un program ar putea crea un șablon de funcție numit *sortare_vector* în scopul sortării elementelor unui vector. În acest caz, funcția ar putea utiliza doi parametri, dintre care primul ar corespunde vectorului, iar cel de-al doilea ar reprezenta numărul elementelor din vector. Dacă presupunem că vectorul nu va reține niciodată mai mult de 32.767 de valori, programul va putea folosi un parametru de tip *int* pentru dimensiunea vectorului. Un șablon mai general va permite, însă, programului să specifice propriul său tip pentru acest parametru, ca mai jos:

```
template<class T, class T1> void sortare_vector
(T vector[], T1 elemente)
  {
    // instructiuni
  }
```

Utilizând șablonul *sortare_vector*, programul poate crea funcții care sortează un vector mai mic (sub 128 de elemente) de tip *float* și un vector foarte mare de tip *int* prin intermediul următoarelor prototipuri:

```
void sortare_vector(float, char);
void sortare_vector(int, long);
```

Ce trebuie să știți

Utilizarea șabloanelor de funcții reduce efortul de programare prin însărcinarea compilatorului de C++ cu generarea instrucțiunilor acelor funcții care diferă numai prin tipul întors sau prin tipurile parametrilor. În lecția 32, „Utilizarea șabloanelor de clase", veți vedea cum puteți folosi șabloane pentru a crea clase fără tipuri, sau generice. Dar înainte de a trece la lecția 32, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Șabloanele de funcții vă permit să declarați o funcție fără tipuri, sau generică.
- ☑ Atunci când trebuie să folosească o funcție cu anumite tipuri de date, programul specifică un prototip de funcție care definește tipurile corespunzătoare.
- ☑ Atunci când întâlnește prototipul funcției, compilatorul de C++ va genera instrucțiunile corespunzătoare ale funcției, ținând cont de tipurile precizate.
- ☑ Programele ar trebui să utilizeze șabloane pentru acele funcții uzuale care lucrează cu tipuri diverse. Cu alte cuvinte, în cazul unei funcții care lucrează cu un singur tip nu se impune utilizarea unui șablon.
- ☑ Dacă o funcție implică mai multe tipuri, șablonul asociază fiecărui tip câte un identificator unic, precum *T*, *T1* sau *T2*. Ulterior, la compilare, compilatorul de C++ va atribui acele tipuri pe care le specificați în prototipul funcției.

Lecția 32
Utilizarea șabloanelor de clase

În lecția 31, „Utilizarea șabloanelor de funcții", ați învățat să folosiți șabloanele de funcții din C++ pentru a crea funcții generice, sau fără tip. Prin definirea șabloanelor de funcții puteați determina ulterior compilatorul de C++ să creeze funcții în locul dumneavoastră, funcții care se deosebeau prin tipurile valorii întoarse și a parametrilor. Așa cum este posibilă crearea de funcții similare care diferă doar prin tipuri, tot astfel pot exista cazuri în care programele să necesite crearea de clase generice. În acest scop, programele pot defini *șabloane de clase*. Lecția de față studiază pașii care trebuie efectuați într-un program pentru a defini și utiliza șabloane de clase. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Prin specificarea cuvântului cheie *template* și a simbolurilor de tip, precum *T*, *T1* sau *T2*, programele pot crea un șablon de clasă – șablonul definiției de clasă poate utiliza aceste simboluri pentru a indica date membru, valori întoarse și tipuri de parametri pentru funcțiile membru, etc.
- Pentru a crea un obiect pe baza unui șablon de clasă, este suficient să menționați în program numele clasei urmat de tipurile pe care compilatorul le va atribui fiecărui simbol. În program se specifică tipurile între paranteze unghiulare (cum ar fi *<int, float>*) și numele variabilei.
- Dacă clasa oferă o funcție constructor cu ajutorul căreia să inițializați variabilele membru, atunci puteți apela acest constructor la crearea unui obiect pe baza unui șablon, ca de pildă *nume_clasa<int, float> valori(200);*.
- Atunci când întâlnește declarația obiectului, compilatorul de C++ creează pe baza șablonului respectiv o clasă ce utilizează tipurile de date corespunzătoare.

Ca și în cazul șabloanelor de funcții, v-ați putea lăsa intimidat la „prima vedere" de aceste șabloane de clase. Cu toate acestea, după crearea și utilizarea a unul sau două șabloane de clase, vă veți convinge că acestea sunt foarte explicite și ușor de folosit.

***Notă:** Compilatorul **Turbo C++ Lite** oferit pe CD-ROM-ul ce însoțește această carte nu oferă suport pentru utilizarea șabloanele din C++.*

Crearea unui șablon de clasă

Ca exemplu de utilizare a șabloanelor de clasă, să presupunem că creați o clasă *array* care oferă metode ce calculează suma și media valorilor reținute în vector. Dacă elementele din vector ar avea tipul *int*, clasa descrisă ar putea fi următoarea:

```
class array {
  public:
    array(int size);
    long sum(void);
    int average_value(void);
    void show_array(void);
    int add_value(int);
  private:
    int *data;
    int size;
    int index;
};
```

Programul următor, *I_Array.CPP*, folosește clasa *array* pentru a lucra cu valori de tip *int*:

```
#include <iostream.h>
#include <stdlib.h>

class array {
  public:
    array(int size);
    long sum(void);
    int average_value(void);
    void show_array(void);
    int add_value(int);
  private:
    int *data;
    int size;
    int index;
  };

array::array(int size)
  {
    data = new int[size];

    if (data == NULL)
    {
      cerr << "Insufficient memory—program ending" << endl;
```

```
      exit(1);
    }
    array::size = size;
    array::index = 0;
  }

long array::sum(void)
  {
    long sum = 0;
    for (int i = 0; i < index; i++)
      sum += data[i];
    return(sum);
  }

int array::average_value(void)
  {
    long sum = 0;
    for (int i = 0; i < index; i++)
      sum += data[i];
    return(sum / index);
  }

void array::show_array(void)
  {
    for (int i = 0; i < index; i++)
      cout << data[i] << ' ';
    cout << endl;
  }

int array::add_value(int value)
  {
    if (index == size)
      return(-1); // Vectorul este plin
    else
    {
      data[index] = value;
```

```
         index++;
         return(0); // Succes
      }
   }

void main(void)
   {
      array numbers(100); // Vector cu 100 de elemente
      int i;

      for (i = 0; i < 50; i++)
         numbers.add_value(i);

      numbers.show_array();

      cout << "The sum of the numbers is " << numbers.sum() <<
        endl;
      cout << "The average value is " <<
        numbers.average_value() << endl;
   }
```

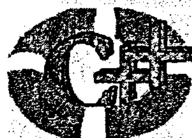
După cum puteți vedea, programul *I_Array.CPP* alocă un vector de 100 de elemente și apoi atribuie valori pentru 50 de elemente ale vectorului cu ajutorul metodei *add_value*. În cadrul clasei *array*, membrul *index* reține numărul de elemente aflate curent în vector. Dacă programul încearcă să adauge mai multe elemente decât poate stoca vectorul, atunci funcția *add_value* întoarce o eroare. Precum vedeți, funcția *average_value* folosește membrul *index* pentru a calcula valoarea medie a elementelor vectorului. Alocarea memoriei pentru vector se face în program prin intermediul operatorului *new*, despre care va discuta în detaliu lecția 33, „Utilizarea memoriei de date în C++".

Pentru un al doilea exemplu de utilizare a șabloanelor de clase, să presupunem că programul trebuie să lucreze acum cu un vector de valori în virgulă mobilă, pe lângă vectorul de întregi. O soluție în oferirea suportului pentru tipurile diferite de vectori este crearea unor clase separate. Prin utilizarea șabloanelor de clase, însă, puteți evita duplicarea unei clase. Următorul șablon de clasă creează clasa generică *array*:

```
template<class T, class T1>
class array {
   public:
      array(int size);
      T1 sum(void);
```

```
      T average_value(void);
      void show_array(void);
      int add_value(T);
   private:
      T *data;
      int size;
      int index;
   };
```

Despre șabloanele de clase



Pe măsură ce numărul claselor create va crește, puteți descoperi că o clasă scrisă pentru un program este foarte similară cu clasa de care aveți nevoie pentru programul la care tocmai lucrați. De multe ori, clasele se deosebesc numai prin tipuri. Cu alte cuvinte, s-ar putea ca o clasă să lucreze cu valori întregi, în timp ce programul dumneavoastră necesită prelucrarea de valori de tip *float*. Pentru a spori capacitatea de refolosire a codului existent, C++ permite programelor să definească șabloane de clase. Pe scurt, un șablon de clasă definește o clasă fără tipuri (sau generică) pe baza căreia puteți crea ulterior obiecte ce folosesc tipuri specifice. Atunci când întâlnește în program o declarație de obiect făcută pe baza unui șablon de clasă, compilatorul de C++ folosește tipurile specificate în declarație pentru a construi clasa corespunzătoare. Dându-vă posibilitatea de a crea rapid clase care diferă numai prin tipuri, șabloanele de clase reduc efortul de programare, economisind astfel timpul dumneavoastră.

Instrucțiunea *template* definește simbolurile de tip *T* și *T1*. În cazul unui vector de valori întregi, *T* va corespunde la *int* și *T1* la *long*. Similar, pentru un vector de valori în virgulă mobilă, *T* și *T1* corespund ambele tipului *float*. Opriți-vă câteva minute pentru a vă asigura că ați înțeles modul în care compilatorul de C++ înlocuiește ulterior simbolurile *T* și *T1* cu tipurile specificate în program.

Înaintea fiecărei funcții a clasei trebuie specificată aceeași declarație *template*. În plus, imediat după numele clasei trebuie precizate simbolurile de tip, ca în *array<T, T1>::average_value*. Instrucțiunile următoare, de exemplu, înfățișează definiția funcției *average_value* din clasa *array*:

```
template<class T, class T1>
T array<T, T1>::average_value(void)
   {
      T1 sum = 0;
```

```
      for (int i = 0; i < index; i++)
         sum += data[i];

      return(sum / index);
   }
```

După definirea şablonului, puteţi crea un obiect corespunzător prin specificarea numelui clasei, urmat de tipurile dorite, plasate între paranteze unghiulare, ca mai jos:

```
array<int, long> numbers(100);
array<float, float> values(200);
```

Numele şablonului

Tipurile pentru şablon

Programul *GenArray.CPP* foloseşte şablonul de clasă *array* pentru a crea două clase: una care lucrează cu valori de tip *int* şi una care lucrează cu valori de tip *float*:

```
#include <iostream.h>
#include <stdlib.h>

template<class T, class T1>
class array {
  public:
     array(int size);
     T1 sum(void);
     T average_value(void);
     void show_array(void);
     int add_value(T);
  private:
     T *data;
     int size;
     int index;
  };

template<class T, class T1>
array<T, T1>::array(int size)
  {
     data = new T[size];

     if (data == NULL)
     {
        cerr << "Insufficient memory—program ending" << endl;
```

```
        exit(1);
      }

      array::size = size;
      array::index = 0;
   }

template<class T, class T1>
T1 array<T, T1>::sum(void)
   {
      T1 sum = 0;

      for (int i = 0; i < index; i++)
         sum += data[i];

      return(sum);
   }

template<class T, class T1>
T array<T, T1>::average_value(void)
   {
      T1 sum = 0;

      for (int i = 0; i < index; i++)
         sum += data[i];

      return(sum / index);
   }

template<class T, class T1>
void array<T, T1>::show_array(void)
   {
      for (int i = 0; i < index; i++)
         cout << data[i] << ' ';

      cout << endl;
   }

template<class T, class T1>
int array<T, T1>::add_value(T value)

   {
      if (index == size)
```

```
      return(-1); // Vectorul este plin
    else
    {
      data[index] = value;
      index++;
      return(0); // Succes
    }
  }

void main(void)
  {
    // Vector cu 100 de elemente
    array<int, long> numbers(100);
    // Vector cu 200 de elemente
    array<float, float> values(200);
    int i;

    for (i = 0; i < 50; i++)
      numbers.add_value(i);

    numbers.show_array();
    cout << "The sum of the numbers is " << numbers.sum() <<
      endl;
    cout << "The average value is " <<
       numbers.average_value() << endl;
    for (i = 0; i < 100; i++)

      values.add_value(i * 100);

    values.show_array();

    cout << "The sum of the numbers is " << values.sum() <<
      endl;
    cout << "The average value is " <<
      values.average_value() << endl;
}
```

Cea mai bună metodă de a înțelege șabloanele de clasă este de a tipări două copii ale acestui program. În primul exemplar înlocuiți fiecare *T* și *T1* cu *int* și *long*, iar în al doilea înlocuiți *T* și *T1* cu *float*.

Declararea obiectelor pe baza unui șablon de clasă



Pentru a crea obiecte pe baza unui șablon de clasă, specificați numele șablonului de clasă, urmat de paranteze unghiulare care să încadreze tipurile de date cu care doriți să fie înlocuite de către compilator simbolurile *T*, *T1*, *T2*, etc. În continuare precizați numele obiectului (variabilei) împreună cu valorile de parametri pe care vreți să le transmiteți funcției constructor, ca mai jos:

```
nume sablon clasa<tip1, tip2> nume obiect(parametru1,
  parametru2);
```

Atunci când întâlnește declarația, compilatorul de C++ creează o clasă pe baza tipurilor specificate în program. Instrucțiunea următoare, de pildă, folosește șablonul de clasă *array* pentru a crea un vector de tip *char* ce poate reține 100 de elemente:

```
array<char, int> numere_mici(100);
```

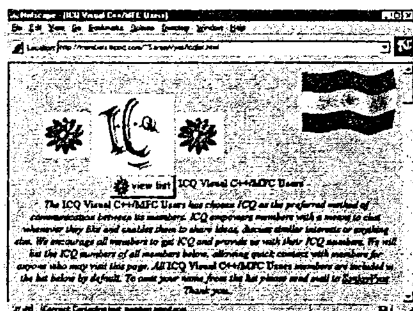
Ce trebuie să știți

În lecția de față ați văzut cum șabloanele de clasă vă pot ajuta la evitarea duplicării de cod în acele situații în care aveți nevoie de obiecte ale unor clase similare, dar care diferă prin anumite tipuri. Deoarece șabloanele de clasă pot fi destul de complexe, este posibil să provoace o ușoară intimidare. Atunci când definiți o clasă, începeți definiția ca și cum ați crea acea clasă pentru tipuri deja cunoscute. După definirea întregii clase, identificați acei membri care trebuie modificați pentru a accepta tipuri diferite. Apoi înlocuiți tipurile membrilor respectivi cu simboluri precum *T*, *T1*, *T2* și așa mai departe.

Programele prezentate în această lecție au utilizat operatorul *new* din C++ pentru a aloca memorie dinamic (la execuția programului) în scopul stocării vectorilor. În lecția 33, „Utilizarea memoriei de date în C++", vom discuta în detaliu despre operatorul *new*. Dar înainte de a trece la lecția 33, asigurați-vă că ați reținut următoarele aspecte importante:

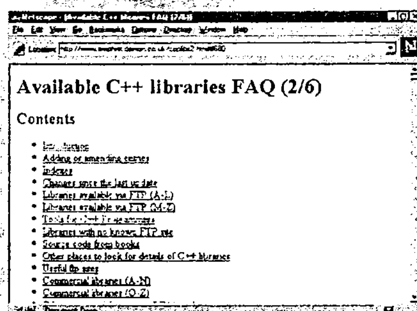
- ☑ Șabloanele de clase vă permit eliminarea codului duplicat pentru acele clase ale căror obiecte diferă numai prin tipurile membrilor.
- ☑ Pentru crearea unui șablon de clasă, precedați definiția clasei cu cuvântul cheie *template* și simboluri de tip precum *T* sau *T1*.
- ☑ Fiecare funcție a clasei trebuie precedată de aceeași instrucțiune *template*. În plus, între numele clasei și operatorul de rezoluție trebuie specificate tipurile șablonului, încadrate de paranteze unghiulare, sub forma *nume_clasa<T, T1>::nume_functie*.
- ☑ Pentru a crea un obiect pe baza unui șablon de clasă, specificați numele șablonului, urmat de paranteze unghiulare care să încadreze tipurile de substituit, ca de pildă *nume_clasa<int, long> obiect*.

UTILIZATORI ICQ VISUAL C++/MFC



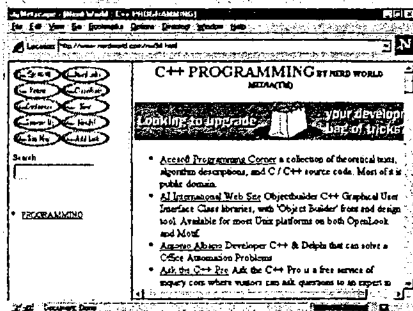
http://member.tripod.com/~SanjayVyas/icqlist.html

BIBLIOTECI C++



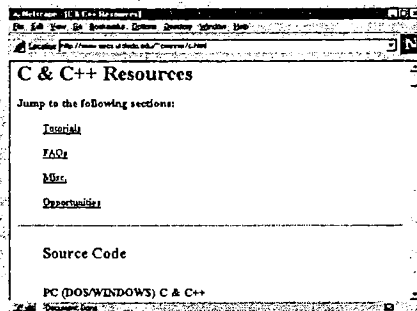
http://www.trmpbrst.demon.co.uk/cpplibs2.html#680

PROGRAMARE C++



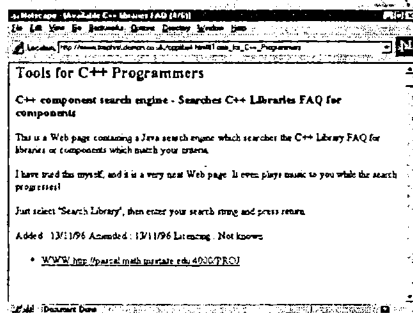
http://www.nerdworld.com/nw94.html

RESURSE C&C++



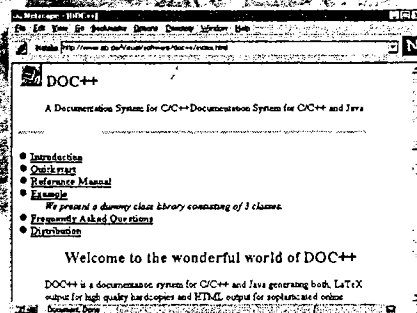
http://www.eecs.utoledo.edu/~cwinner/c.html

INSTRUMENTE PENTRU PROGRAMATORI C++



http://www.trmpbrst.demon.co.uk/cpplibs4.html#Tools_for_C++_Programmers

DOCUMENTAȚIE PENTRU C/C++



http://www.zib.de/Visual/software/doc++/index.html

PARTEA a VI-a

Elemente avansate de C++

Primele cinci părţi ale acestei cărţi au prezentat acele elemente ce privesc C++ şi programarea orientată spre obiect, pe care trebuie să le cunoaşteţi pentru a putea crea programe puternice. În această parte vă veţi împlini cunoştinţele de C++, învăţând să alocaţi memorie în timpul execuţiei unui program. Veţi învăţa, de asemenea, să efectuaţi operaţii cu fişiere în C++ şi să utilizaţi excepţiile din C++ pentru a răspunde la apariţia de erori neprevăzute în program. În parcurgerea lecţiilor din partea de faţă nu vă lăsaţi intimidat de titlul acesteia, „Elemente avansate de C++". După cum veţi vedea, aceste elemente nu vor fi mai dificile decât cele pe care aţi ajuns deja să le stăpâniţi pe parcursul cărţii. Lecţiile aflate în această parte sunt următoarele:

Lecția 33

Utilizarea memoriei de date în C++

Așa cum ați văzut, atunci când un program declară un vector, compilatorul de C++ alocă memorie pentru a reține elementele vectorului respectiv. Cu timpul, este posibil ca dimensiunea vectorului să devină insuficientă pentru păstrarea tuturor datelor. Spre exemplu, să presupunem că definiți un vector care poate reține prețurile pentru 100 de acțiuni. Dacă ulterior va trebui să păstrați prețurile a mai mult de 100 de acțiuni, atunci veți fi nevoit să editați și să recompilați programul. Ca o alternativă la alocarea vectorilor de dimensiune constantă, programele pot aloca *dinamic* cantitatea de memorie necesară în momentul execuției. De pildă, în cazul în care programul are de urmărit prețurile a 100 de acțiuni, acesta ar aloca memoria necesară pentru un vector cu 100 de elemente. Similar, dacă programul trebuie să rețină numai 25 de acțiuni, acesta ar putea aloca mai puțină memorie. Prin alocarea dinamică a memoriei, programele se adaptează continuu nevoilor existente, fără a presupune un efort suplimentar de programare. Atunci când alocă memorie în timpul execuției, programele specifică o cantitate de memorie necesară, iar C++ întoarce un pointer la acea memorie. Regiunea de memorie din care C++ efectuează alocările se numește *memorie de date*. Lecția de față se oprește asupra pașilor ce trebuie efectuați de un program pentru a aloca dinamic și apoi a elibera memorie în timpul execuției. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru alocarea de memorie în timpul execuției, programele C++ apelează la operatorul *new*.
- La utilizarea operatorului *new*, programele precizează cantitatea de memorie necesară, în octeți.
- Dacă operatorul *new* reușește alocarea memoriei solicitate, acesta va întoarce un pointer la începutul regiunii de memorie respective.
- Dacă operatorul *new* nu reușește să satisfacă cererea de memorie a programului (este posibil să nu mai existe memorie liberă), acesta va întoarce un pointer NULL.
- Pentru a elibera ulterior memoria alocată cu *new*, programele utilizează operatorul *delete* din C++.

Alocarea dinamică a memoriei pe parcursul execuției reprezintă o facilitate foarte puternică. Experimentați cu programele prezentate în această lecție. Așa cum veți vedea, alocarea dinamică a memoriei este, de fapt, foarte simplă.

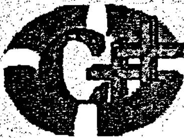
Utilizarea operatorului new

Operatorul *new* din C++ permite programelor să aloce memorie dinamic pe durata execuției. Pentru a utiliza operatorul *new* trebuie să specificați numărul de octeți de care are nevoie programul. Spre exemplu, să presupunem că programul necesită un vector de 50 de octeți. Prin intermediul operatorului *new*, alocarea memoriei se poate face astfel:

```
char *buffer = new char[50];
```

Dacă alocarea memoriei reușește, operatorul *new* întoarce un pointer la începutul zonei de memorie alocate. În cazul de față, din cauză că programul alocă memorie pentru a reține un vector de caractere, programul atribuie adresa unei variabile pointer la tipul *char*. Dacă alocarea memoriei solicitate eșuează, operatorul *new* va întoarce un pointer NULL. La fiecare alocare dinamică de memorie prin intermediul operatorului *new*, programele ar trebui să verifice dacă valoarea întoarsă de *new* este NULL. Dacă această valoare este NULL, atunci *new* nu a reușit să satisfacă cererea de memorie a programului.

De ce alocă programele memorie dinamic prin intermediul operatorului new



Un număr mare de programe folosesc vectori pentru a păstra mai multe valori de un anumit tip. Atunci când scriu codul, programatorii vor încerca în mod normal să declare vectori ale căror dimensiuni să acopere nevoile viitoare ale programului. Din păcate, în cazul în care nevoia de memorie a programului depășește anticipările programatorului, cineva va trebui să editeze și să recompileze întregul program.

Ca o alternativă la editarea și recompilarea programelor numai pentru că acestea au nevoie de mai multă memorie, este de preferat să scrieți programele astfel încât să aloce la rulare atâta memorie câtă este necesară, iar în acest scop puteți apela la operatorul *new*. Astfel, programele pot modifica modul de utilizare a memoriei după cum este nevoie, eliminând necesitatea de a edita și recompila programul.

Ca un exemplu de alocare dinamică a memoriei, programul următor, *Use_New.CPP*, folosește operatorul *new* pentru a aloca un pointer la un vector de 100 de caractere:

```
#include <iostream.h>

void main(void)
  {
    char *pointer = new char[100];

    if (pointer != NULL)
```

```
        cout << "Memory successfully allocated" << endl;
    else
        cout << "Error allocating memory" << endl;
}
```

Precum vedeţi, programul *Use_New.CPP* testează imediat valoarea pe care operatorul *new* a atribuit-o variabilei pointer. Dacă această valoare este NULL, atunci *new* nu a reuşit să aloce memoria solicitată. Dacă pointerul nu conţine valoarea NULL, atunci *new* a alocat cu succes memorie şi valoarea pointerului conţine adresa de început a blocului de memorie.

Dacă nu reuşeşte să aloce memoria solicitată, new întoarce valoarea NULL



Atunci când programele utilizează operatorul *new* pentru a aloca memorie, pot exista situaţii în care C++ nu poate satisface o cerere de memorie deoarece în memoria de date nu mai există suficient spaţiu disponibil. Atunci când nu reuşeşte să aloce memoria solicitată, operatorul *new* atribuie pointerului valoarea NULL. Testând valoarea pointerului, programul poate determina dacă *new* a satisfăcut cererea de memorie. Instrucţiunea următoare, de pildă, foloseşte *new* în scopul alocării de memorie pentru un vector de 500 de numere în virgulă mobilă:

```
float *vector = new float[500];
```

Pentru a afla dacă *new* a reuşit să aloce memorie, programul ar trebui să compare pointerul cu valoarea NULL, ca mai jos:

```
if (vector != NULL)
    cout << "Memoria a fost alocata" << endl;
else
    cout << "new nu a alocat memorie" << endl;
```

Programul anterior a folosit operatorul *new* pentru a aloca 100 de octeţi de memorie. Deoarece cantitatea de memorie solicitată este „bătută în cuie" (specificată explicit în cod), în cazul în care doriţi ca programul să aloce mai multă sau mai puţină memorie sunteţi nevoit să-l editaţi şi să-l recompilaţi. Aşa cum menţionam, unul dintre motivele pentru alocarea dinamică a memoriei este eliminarea necesităţii de a edita şi recompila un program odată cu schimbarea consumului de memorie.

Programul următor, *Ask_Mem.CPP*, solicită utilizatorului să introducă numărul de octeţi necesari programului, după care alocă memoria corespunzătoare cu ajutorul operatorului *new*.

```
#include <iostream.h>

void main(void)
  {
    int size;
    char *pointer;

    cout << "Type in the array size, up to 30000: ";
    cin >> size;

    if (size <= 30000)
    {
      pointer = new char[size];

      if (pointer != NULL)
         cout << "Memory successfully allocated" << endl;
      else
         cout << "Unable to allocate memory" << endl;
    }
  }
```

Atunci când utilizează operatorul *new* pentru a aloca memorie dinamic, programele au, de obicei, o cale de a determina intern cantitatea de memorie care trebuie alocată. Spre exemplu, dacă un program alocă memorie pentru a reține informații despre angajați, acesta ar putea păstra numărul angajaților într-un fișier. La lansare, programul poate citi din fișier numărul de angajați, alocând apoi memorie în mod corespunzător.

Următorul program, *NoMemory.CPP*, rezervă câte 10000 de caractere odată, până când *new* nu mai reușește alocarea de spațiu din memoria de date. Cu alte cuvinte, programul alocă memorie până când ocupă toată memoria de date disponibilă. Atunci când alocarea de memorie reușește, programul afișează un mesaj care informează că a fost alocată memorie. Când alocarea de memorie eșuează, programul afișează un mesaj de eroare și își încheie execuția, așa cum este ilustrat în continuare:

```
#include <iostream.h>

void main(void)
  {
    char *pointer;

    do {
      pointer = new char[10000];
```

```
            if (pointer != NULL)
               cout << "Allocated 10,000 bytes" << endl;
            else
               cout << "Memory allocation failed" << endl;
        } while (pointer != NULL);
    }
```

Notă: În cazul în care compilarea și rularea se fac sub mediul MS-DOS, ați putea fi surprins să descoperiți că memoria de date se consumă după ce spațiul alocat de program depășește 64KB. În mod implicit, majoritatea compilatoarelor de C++ pentru MS-DOS utilizează modelul redus de memorie, care oferă o memorie de date de 64KB. De asemenea, dacă vă aflați în mediul DOS, cea mai mare zonă de memorie pe care programele o pot accesa este limitată la 64KB.

Despre memoria de date



La fiecare execuție a unui program, compilatorul de C++ rezervă o zonă de memorie nefolosită care se numește *memorie de date*. Prin intermediul operatorului *new*, programele pot aloca în timpul execuției spațiu din memoria de date. Utilizarea memoriei de date pentru o astfel de alocare de memorie face ca programele să nu fie limitate la vectori de dimensiuni fixe. În funcție de sistemul de operare și de modelul de memorie folosit de compilator, dimensiunea memoriei de date poate varia. Pe măsură ce cantitatea de memorie dinamică necesară programelor crește, va trebui să țineți cont de limitele impuse de sistem asupra memoriei de date.

Eliberarea memoriei atunci când programul nu mai are nevoie de aceasta

După cum ați învățat, operatorul *new* din C++ permite programelor să aloce dinamic memorie în timpul execuției. Atunci când un program nu mai are nevoie de memoria alocată, acesta ar trebui să elibereze memoria respectivă prin intermediul operatorului *delete*. Eliberarea memoriei cu ajutorul operatorului *delete* se face prin simpla specificare a pointerului la zona de memorie respectivă, ca aici:

```
delete pointer;
```

Programul următor, *Del_Mem.CPP*, folosește operatorul *delete* pentru a elibera memoria alocată cu operatorul *new*.

```
#include <iostream.h>
#include <string.h>

void main(void)
  {
    char *pointer = new char[100];

    strcpy(pointer, "Rescued by C++");

    cout << pointer << endl;

    delete pointer;
  }
```

În mod implicit, dacă un program nu eliberează memoria pe care a alocat-o, atunci sistemul de operare va recupera automat această memorie după încheierea programului. Dacă programele folosesc, însă, operatorul *delete* pentru a elibera memoria imediat ce aceasta nu mai este necesară, respectiva memorie devine disponibilă pentru alte utilizări (spre exemplu, s-ar putea ca programul să folosească ulterior operatorul *new* pentru a aloca memoria în alt scop, sau este posibil ca sistemul de operare să aloce acea memorie pentru utilizarea sa de către un alt program).

Un nou exemplu

Programul următor, *AloccArr.CPP*, alocă memorie pentru a stoca un vector de 1000 de numere întregi. Programul atribuie apoi elementelor vectorului valorile de la 1 la 1000, afișându-le totodată pe ecran. În continuare, memoria este eliberată și se alocă un vector de 2000 de numere în virgulă mobilă, elementelor acestuia fiindu-le atribuite valorile de la 1,0 la 2000,0, așa cum se vede în continuare:

```
#include <iostream.h>

void main(void)
  {
    int *int_array = new int[1000];
    float *float_array;
    int i;

    if (int_array != NULL)
    {
      for (i = 0; i < 1000; i++)
        int_array[i] = i + 1;
```

```
        for (i = 0; i < 1000; i++)
          cout << int_array[i] << ' ';
        delete int_array;
      }
      float_array = new float[2000];
      if (float_array != NULL)
        {
          for (i = 0; i < 2000; i++)
            float_array[i] = (i + 1) * 1.0;
          for (i = 0; i < 2000; i++)
            cout << float_array[i] << ' '
          delete float_array;
        }
}
```

Ca o regulă, este bine ca programele să elibereze memoria prin intermediul operatorului *delete*, imediat ce respectiva memorie își pierde utilitatea.

Ce trebuie să știți

În lecția de față ați învățat că programele pot aloca dinamic memorie în timpul execuției dintr-o zonă de memorie disponibilă care se numește memorie de date, utilizând în acest scop operatorul *new*. Prin această alocare dinamică a memoriei, programele se adaptează nevoilor curente, fără a mai fi necesară editarea și recompilarea lor. În lecția 34, „Controlul operațiilor cu memoria de date", veți vedea cum puteți să controlați alocarea memoriei de date și operațiile pe care *new* le efectuează atunci când nu poate satisface o cerere de memorie. Dar, înainte de a trece la lecția 34, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Capacitatea unui program de a aloca memorie dinamic, în timpul execuției, elimină dependența programului de vectorii de dimensiuni fixe.
- ☑ Atunci când reușește alocarea memoriei solicitate de program, operatorul *new* întoarce un pointer către începutul zonei de memorie respective.
- ☑ În cazul în care nu reușește să aloce memoria solicitată de program, operatorul *new* întoarce un pointer NULL.

- ☑ De fiecare dată când alocă dinamic memorie prin intermediul lui *new*, programele trebuie să verifice dacă valoarea pointerului întors de *new* nu este NULL, ceea ce ar indica eșecul operației de alocare a memoriei.
- ☑ Programele pot accesa memoria alocată cu operatorul *new* prin intermediul unui pointer sau al unui vector.
- ☑ Operatorul *new* alocă spațiu dintr-o zonă de memorie nefolosită numită memorie de date.
- ☑ Dimensiunea memoriei de date diferă în funcție de sistemul de operare și de modelul de memorie folosit de compilator. Mediul MS-DOS, de pildă, poate limita memoria de date la 64KB.
- ☑ Atunci când nu mai au nevoie de memoria alocată, programele ar trebui să elibereze acea memorie cu ajutorul operatorului *delete*.

Lecția 34

Controlul operațiilor cu memoria de date

În lecția 33, „Utilizarea memoriei de date în C++", ați învățat că, pe durata execuției, programele pot utiliza operatorul *new* pentru a aloca dinamic spațiu din memoria de date. Dacă operatorul *new* reușește alocarea memoriei, programul va primi un pointer la zona de memorie respectivă. Dacă *new* nu poate satisface cererea de memorie a programului, atunci va întoarce valoarea NULL. În funcție de scopul programului, pot exista cazuri în care să doriți efectuarea unor anumite operații atunci când *new* nu poate satisface o cerere de memorie. În lecția de față veți învăța să instruiți C++ să apeleze o funcție anume din program atunci când *new* eșuează într-o alocare de memorie. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- În cadrul unui program, puteți să definiți propriul *sistem de tratare a cazurilor de memorie insuficientă* (o funcție), la care C++ va apela atunci când *new* nu poate satisface o cerere de memorie.
- C++ vă permite să definiți propriul operator *new* pentru alocarea și, eventual, inițializarea memoriei.
- C++ vă permite să definiți propriul operator *delete* pentru eliberarea de memorie.

Așa cum veți vedea, prin crearea de operatori *new* și *delete* proprii veți putea controla mai bine modul în care programul tratează erorile de lipsă de memorie.

Crearea unui sistem de tratare a cazurilor de memorie insuficientă

Așa cum ați aflat din lecția 33, atunci când nu poate aloca spațiu din memoria de date, operatorul *new* atribuie variabilei pointer valoarea NULL. Următorul program, *Use_Free.CPP*, utilizează repetat operatorul *new*, alocând câte 1000 de octeți până când memoria de date devine ocupată:

```
#include <iostream.h>

void main(void)
 {
   char *pointer;

   do {
     pointer = new char[1000];
```

```
        if (pointer != NULL)
           cout << "Allocated 1000 bytes" << endl;
        else
           cout << "Free store empty" << endl;

      } while (pointer);
   }
```

După cum puteți vedea, programul *Use_Free.CPP* iterează până când operatorul *new* atribuie pointerului valoarea NULL. Dacă vreți ca *new* să efectueze alte operații atunci când nu poate satisface o cerere de memorie (altceva decât simpla întoarcere a valorii NULL), definiți mai întâi o funcție pe care doriți ca programul să o apeleze când nu există suficientă memorie pentru o cerere. Spre exemplu, funcția următoare, *end_program*, afișează pe ecran un mesaj și apoi folosește funcția *exit* din biblioteca de execuție pentru a încheia programul:

```
void end_program(void)
  {
    cout << "Memory request cannot be satisfied" << endl;
    exit(1);
  }
```

Pentru a instrui C++ să apeleze funcția *end_program* atunci când *new* nu poate satisface o cerere de memorie, apelați funcția *set_new_handler* având ca parametru funcția *end_program*, ca în continuare:

```
set_new_handler(end_program);
```

Programul următor, *End_Free.CPP*, apelează funcția *end_program* atunci când operatorul *new* nu reușește satisfacerea unei cereri de memorie:

```
#include <iostream.h>
#include <stdlib.h> // Prototipul functiei exit
#include <new.h>    // Prototipul functiei set_new_handler

void end_program(void)
  {
    cout << "Memory request cannot be satisfied" << endl;
    exit(1);
  }
```

```
void main(void)
   {
      char *pointer;
      set_new_handler(end_program);
      do {
         pointer = new char[10000];
         cout << "Allocated 10000 bytes" << endl;
      } while (1);
   }
```

În acest exemplu, dacă operatorul *new* nu reușește să aloce spațiu din memoria de date, programul își încheie pur și simplu execuția. În funcție de necesitățile programului, ați putea utiliza o funcție care să aloce memorie dintr-o altă sursă, cum ar fi memoria extinsă a calculatorului care există în mediul MS-DOS. În plus, programul ar putea decide să elibereze memoria alocată anterior pentru alte scopuri, sporind astfel memoria de date disponibilă. Oferind programelor posibilitatea de a trata într-un mod propriu situațiile de lipsă de memorie, C++ vă permite un control total asupra procesului de alocare a memoriei.

***Notă:** Compilatorul **Turbo C++ Lite** aflat pe CD-ROM-ul care însoțește această carte nu acceptă utilizarea funcției **set_new_handler**.*

Definirea proprriilor operatori new și delete

Așa cum ați văzut, C++ permite programelor supradefinirea de operatori. În acest fel puteți supradefini, de asemenea, operatorii *new* și *delete* pentru a le altera comportamentul. De exemplu, să presupunem că alocați 100 de octeți de memorie pentru a reține datele super-secrete ale firmei. Când eliberați mai târziu memoria cu ajutorul operatorului *delete*, programul va marca zona care conținea datele ca fiind liberă. Dacă un spion formidabil (și programator de C++) are acces la calculatorul dumneavoastră, acesta ar putea scrie, teoretic, ca programator, un program care găsește vectorul de 100 de octeți din memoria calculatorului și afișează super-secretele pe ecran.

Prin supradefinirea operatorului *delete*, programul poate să umple mai întâi zona de memorie cu zero sau cu caractere aleatoare și apoi să o elibereze. Următorul program, *MyDelete.CPP*, supradefinește operatorul *delete*. Programul suprascrie cei 100 de octeți indicați de pointer, după care eliberează memoria cu ajutorul funcției *free* din biblioteca de execuție:

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

static void operator delete(void *pointer)
  {
    char *data = (char *) pointer;
    int i;

    for (i = 0; i < 100; i++)
      data[i] = 0;

    cout << "The secrets are safe!" << endl;

    free(pointer);
  }

void main(void)
  {
    char *pointer = new char[100];

    strcpy(pointer, "My Company Secrets");

    delete pointer;
  }
```

La lansare, programul alocă memorie pentru un vector șir de caractere apelând la *new*. În acest șir sunt copiate apoi secretele firmei. În fine, programul folosește operatorul *delete* pentru a elibera memoria. În cadrul funcției *delete*, instrucțiunea următoare atribuie pointerul primit unui pointer la șir de caractere:

```
char *data = (char *) pointer;
```

Șirul *(char *)* reprezintă o *conversie de tip* al cărei unic rol este să arate compilatorului C++ că funcția este conștientă de faptul că un pointer de tip *void* (vedeți parametrul funcției de mai sus) este atribuit unui pointer de tip *char*. Dacă înlăturați această conversie, compilarea programului va eșua. Funcția înscrie apoi valoarea zero în cei 100 de octeți ai zonei de memorie și eliberează memoria folosind funcția *free* din biblioteca de execuție. Este important să remarcați că funcția noastră este valabilă numai pentru o regiune de memorie de 100 de octeți. Deoarece programul *MyDelete.CPP* alocă memorie o singură dată, funcția *free* își face datoria și programul se execută corect. Dacă modificați programul astfel încât să aloce numai 10 octeți de memorie și nu faceți modificările corespunzătoare în funcția *delete*, aceasta va suprascrie 90 de octeți de memorie pe care programul i-ar putea folosi în alt scop, cauzând astfel erori. Programele pot afla, însă, prin intermediul

funcţiilor din biblioteca de execuţie, informaţii despre dimensiunea zonei de memorie indicate de un pointer.

Programul anterior, *MyDelete.CPP*, a supradefinit operatorul *delete*. Într-un mod similar, următorul program, *New_Over.CPP*, supradefineşte operatorul *new* din C++. În acest caz, operatorul supradefinit plasează şirul „Rescued by C++!" la începutul zonei de memorie alocate:

```
#include <iostream.h>
#include <alloc.h>
#include <string.h>

static void *operator new(size t_size)
  {
    char *pointer;

    pointer = (char *) malloc(size);

    if (size > strlen("Rescued by C++!"));
      strcpy(pointer, "Rescued by C++!");

    return(pointer);
  }

void main(void)
  {
    char *str = new char[100];

    cout << str << endl;
  }
```

Aşa cum se vede, funcţia *new* foloseşte pentru alocarea memoriei funcţia *malloc* din biblioteca de execuţie. Dacă dimensiunea memoriei alocate este suficient de mare pentru a reţine şirul „Rescued by C++!", atunci funcţia *new* foloseşte funcţia *strcpy* din biblioteca de execuţie pentru a copia şirul în zona de memorie. Prin supradefinirea operatorilor *new* şi *delete*, programele pot controla mult mai bine operaţiile de alocare a memoriei.

Ce trebuie să ştiţi

Pe măsură ce cresc în complexitate, programele pe care le creaţi vor aloca memorie în timpul execuţiei prin intermediul operatorului *new*. În lecţia de faţă aţi învăţat să modificaţi comportamentul operatorului *new*, mai întâi prin definirea unei funcţii de tratare pe care programul o apelează atunci când *new* nu poate satisface o cerere de memorie, iar apoi prin supradefinirea operatorului *new* însuşi. În lecţia 35, „Alte operaţii cu cin şi cout", veţi învăţa despre noi moduri de utilizare a fluxurilor de intrare şi de ieşire *cin* şi, respectiv,

cout în scopul îmbunătățirii operațiilor de intrare și ieșire din cadrul programelor. Dar, înainte de a trece la lecția 35, asigurați-vă că ați reținut următoarele aspecte importante:

☑ În mod implicit, atunci când nu poate satisface o cerere de memorie, operatorul *new* întoarce valoarea NULL.

☑ Dacă este necesară efectuarea unor alte operații atunci când *new* nu poate satisface o cerere de memorie, programele pot defini propriile sisteme de tratare a acestor cazuri. Prin intermediul funcției *set_new_handler*, un program poate instrui *new* să apeleze o anumită funcție atunci când nu reușește o alocare de memorie.

☑ C++ permite programelor supradefinirea operatorilor *new* și *delete*. Dar, înainte de a apela la această facilitate, asigurați-vă că aveți o bună înțelegere a memoriei de date și a funcțiilor din biblioteca de execuție care lucrează cu aceasta.

Lecția 35
Alte operații cu cin și cout

Pe întreg parcursul acestei cărți ați folosit fluxul de ieșire *cout* pentru a afișa informații pe ecran. De asemenea, multe din programe au apelat la fluxul de intrare *cin* pentru a citi informații de la tastatură. Dacă studiați fișierul de antet *iostream.h*, puteți observa că acest fișier definește *cin* și *cout* ca și obiecte de clasă. Ca obiecte, *cin* și *cout* acceptă diferiți operatori și diverse operații. În lecția de față veți învăța să îmbunătățiți operațiile de intrare și ieșire din cadrul programului prin intermediul metodelor conținute în clasele *cin* și *cout*. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Fișierul de antet *iostream.h* conține definițiile de clasă, pe care le puteți studia pentru o mai bună înțelegere, a fluxurilor de intrare/ieșire (I/E).
- Prin intermediul metodei *cout.width*, programele pot controla lățimea de ieșire.
- Prin intermediul metodei *cout.fill*, programele pot înlocui spațiile albe de la ieșire (tabulatori și spații) cu un caracter anume.
- Pentru a controla numărul de cifre după virgulă afișate de *cout*, programele pot apela la metoda *cout.setprecision*.
- Pentru a afișa un singur caracter odată sau pentru a citi câte un singur caracter, programele pot folosi metodele *cout.put* și, respectiv, *cin.get*.
- Prin intermediul metodei *cin.getline*, programele pot citi o întreagă linie odată.

Aproape fiecare program C++ pe care îl veți crea va folosi *cout* sau *cin* pentru efectuarea operațiilor I/E. Luați-vă timpul necesar pentru a experimenta cu programele prezentate în această lecție.

O privire asupra fișierului iostream.h

Începând cu lecția 2, „Crearea primului program", toate programele C++ pe care le-ați scris au inclus fișierul de antet *iostream.h*. Dacă examinați conținutul acestui fișier, veți găsi aici definițiile care permit programelor să folosească pe *cout* pentru operații de ieșire și pe *cin* pentru operații de intrare. Mai precis, acest fișier definește clasele *istream* și *ostream* (flux de intrare și flux de ieșire) de care aparțin variabilele obiect *cin* și, respectiv, *cout*.

Tipăriți acum o copie a fișierului *iostream.h*. Acest fișier se află în subdirectorul *INCLUDE* din locația de instalare a compilatorului. Definițiile aflate aici sunt destul de complexe. Dacă parcurgeți, totuși, fișierul pas cu pas, veți vedea că majoritatea definițiilor creează, pur și simplu clase și constante. În acest fișier veți găsi declarațiile ambelor obiecte, *cin* și *cout*.

Utilizarea lui cout

Așa cum ați aflat, *cout* este un obiect care pune la dispoziție mai multe metode. Programul următor ilustrează utilizarea mai multor metode *cout* în scopul formatării ieșirii. După cum ați învățat în lecția 4, „Afișarea mesajelor pe ecran", modificatorul *setw* permite programelor să specifice numărul minim de caractere pe care le va ocupa următoarea valoare de ieșire, ca mai jos:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
  {
     cout << "My favorite number is" << setw(3) << 1001 <<
       endl;
     cout << "My favorite number is" << setw(4) << 1001 <<
       endl;
     cout << "My favorite number is" << setw(5) << 1001 <<
       endl;
     cout << "My favorite number is" << setw(6) << 1001 <<
       endl;
  }
```

Asemeni modificatorului *setw*, metoda *cout.width* permite specificarea numărului minim de caractere pe care *cout* le va folosi pentru a afișa valoarea care urmează apelului de funcție. Programul următor, *CoutWidt.CPP*, folosește metoda *cout.width* pentru a obține același efect cu cel pe care l-am obținut mai devreme prin *setw*:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
  {
     int i

     for (i = 3; i < 7; i++)
     {
        cout << "My favorite number is";
        cout.width(i);
```

```
        cout << 1001 << endl;
      }
    }
```

După compilarea și rularea programului *CoutWidt.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> CoutWidt <Enter>
My favorite number is1001
My favorite number is1001
My favorite number is 1001
My favorite number is  1001
```

Ca și în cazul modificatorului *setw*, lățimea selectată de program prin metoda *cout.width* este valabilă numai pentru următoarea valoare afișată.

Utilizarea repetată a unui caracter

Atunci când utilizați modificatorul *setw* sau metoda *cout.width* pentru a controla lățimea de ieșire, *cout* va plasa înainte de valori (sau după, în cazul alinierii la stânga) atâtea spații câte sunt necesare. În funcție de program, pot exista situații în care să doriți folosirea unui caracter, altul decât spațiu. De exemplu, să presupunem că programul creează un cuprins asemeni următorului:

```
Table of Information
Company Profile.......................................10
Company Profit and Loss...............................11
Company Board Members.................................13
```

În acest exemplu, numerele de pagină afișate sunt precedate de puncte. Metoda *cout.fill* vă permite să precizați caracterul pe care *cout* îl va utiliza pentru a umple spațiile libere. Următorul program, *CoutFill.CPP*, creează un cuprins similar cu cel înfățișat mai sus:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
  {
    cout << "Table of Information" << endl;
    cout.fill('.');
    cout << "Company Profile" << setw(20) << 10 << endl;
    cout << "Company Profit and Loss" << setw(12) << 11 <<
      endl;
```

```
        cout << "Company Board Members" << setw(14) << 13 <<
          endl;
    }
```

După utilizarea metodei *cout.fill* pentru selectarea caracterului de umplere, alegerea făcută rămâne în vigoare până la următorul apel *cout.fill*.

Controlarea cifrelor de după virgulă

În mod implicit, atunci când afișați o valoare în virgulă mobilă cu ajutorul *cout* nu puteți ști niciodată cu certitudine câte cifre vor apărea pe ecran. Prin folosirea modificatorului *setprecision*, însă, este posibilă specificarea numărului de cifre dorit. Programul următor, *SetPrec.CPP*, folosește modificatorul *setprecision* pentru a controla numărul de cifre care apar după virgulă:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
  {
    float value = 1.23456;

    int i;

    for (i = 1; i < 6; i++)
      cout << setprecision(i) << value << endl;
  }
```

După compilarea și rularea programului *SetPrec.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> SetPrec <Enter>
1.2
1.23
1.235
1.2346
1.23456
```

Atunci când utilizați modificatorul *setprecision* pentru modificarea preciziei de afișare, alegerea făcută rămâne valabilă până la o nouă utilizare a modificatorului.

Afişarea unui singur caracter

În funcţie de program, s-ar putea ca uneori să trebuiască să afişaţi un caracter odată sau să citiţi de la tastatură un singur caracter. Pentru a afişa un singur caracter, programele pot apela la metoda *cout.put*. Programul următor, *CoutPut.CPP*, foloseşte această metodă pentru a afişa pe ecran mesajul *Rescued by C++!*, caracter cu caracter:

```
#include <iostream.h>

void main(void)
  {
    char string[] = "Rescued by C++!";
    int i;

    for (i = 0; string[i]; i++)
      cout.put(string[i]);
  }
```

Biblioteca de execuţie pune la dispoziţie o funcţie numită *toupper*, care întoarce majuscula corespunzătoare unei litere. Programul care urmează, *CoutUppr.CPP*, foloseşte funcţia *toupper* pentru a transforma caracterele în majuscule, afişându-le apoi cu *cout.put*:

```
#include <iostream.h>
#include <ctype.h> // Prototipul functiei toupper

void main(void)
  {
    char string[] = "Rescued by C++!";
    int i;

    for (i = 0; string[i]; i++)
      cout.put(toupper(string[i]));

    cout << endl << "Ending string: " << string << endl;
  }
```

După compilarea şi rularea programului *CoutUppr.CPP*, pe ecran vor fi afişate următoarele:

```
C:\> CoutUppr <Enter>
RESCUED BY C++!
Ending string: Rescued by C++!
```

Citirea de la tastatură a unui singur caracter

Așa cum *cout* oferă metoda *cout.put* pentru afișarea unui caracter, tot astfel *cin* pune la dispoziție metoda *cin.get* care vă permite citirea unui singur caracter. Utilizarea metodei *cin.get* se face prin simpla atribuire a caracterului întors de metodă unei variabile, ca mai jos:

```
litera = cin.get();
```

Programul următor, *Cin_Get.CPP*, afișează un mesaj care vă solicită să răspundeți cu Y sau N. Programul citește apoi repetat caractere cu *cin.get* până când întâlnește un Y sau un N:

```
#include <iostream.h>
#include <ctype.h>
void main(void)
  {
    char letter;
    cout << "Do you want to continue? (Y/N)   ";
    do {
       letter = cin.get()
       // Transformam in majuscule
       letter = toupper(letter)
    } while ((letter != 'Y') && (letter != 'N'))
    cout << endl << "You selected " << letter << endl;
  }
```

Citirea de la tastatură a unei întregi linii

Așa cum ați învățat, atunci când efectuați operații de intrare cu ajutorul lui *cin*, acesta folosește caracterele de spațiere, precum: spațiu, tabulator sau retur de car, pentru a determina unde se termină o valoare și unde începe următoarea. De multe ori, veți dori ca programele să citească într-un șir o linie întreagă. În acest scop puteți apela la metoda *cin.getline*. Pentru a utiliza *cin.getline*, specificați șirul de caractere în care metoda va depune literele citite și un parametru indicând dimensiunea șirului, ca mai jos:

```
cin.getline(sir, 64);
```

Atunci când citeşte caractere de la tastatură, *cin.get* nu va depăşi capacitatea de stocare a şirului. O soluţie elegantă pentru exprimarea dimensiunii şirului este folosirea operatorului *sizeof* din C++, ca aici:

```
cin.getline(sir, sizeof(sir));
```

Dacă modificaţi ulterior dimensiunea şirului, nu va trebui să căutaţi şi să schimbaţi fiecare instrucţiune *cin.get* care apare în program. În schimb, operatorul *sizeof* va preciza dimensiunea corectă a şirului. Programul următor, *GetLine.CPP*, utilizează metoda *cin.getline* pentru a citi de la tastatură o linie de text:

```
#include <iostream.h>
void main(void)
  {
    char string[128]
    cout << "Type line of text and press Enter" << endl;
    cin.getline(string, sizeof(string));
    cout << "You typed: " << string << endl;
  }
```

Atunci când programele citesc caractere de la tastatură, pot apărea cazuri în care trebuie citite caractere până la şi inclusiv o literă anume. După ce programul citeşte acea literă, aţi dori ca operaţia de citire să ia sfârşit. Pentru a efectua o astfel de operaţie, programul poate transmite litera în cauză metodei *cin.getline.* Spre exemplu, următorul apel de funcţie determină *cin.getline* să citească o linie de text, încheind operaţia atunci când a citit un Enter, 64 de caractere sau litera Z:

```
cin.getline(string, 64, 'Z');
```

Următorul program, *Until_Z.CPP*, foloseşte metoda *cin.getline* pentru a citi o linie de text sau caractere până la şi inclusiv prima literă Z:

```
#include <iostream.h>
void main(void)
  {
    char string[128];
    cout << "Type line of text and press Enter" << endl;
```

```
    cin.getline(string, sizeof(string), 'Z');
    cout << "You typed: " << string << endl;
}
```

Compilați și rulați programul *Until_Z.CPP*. Experimentați prin introducerea de diferite linii de text, printre care unele care încep cu litera Z, altele care se termină cu litera Z și altele care nu conțin nici un Z.

Ce trebuie să știți

Fiecare program C++ pe care îl scrieți va folosi, probabil, *cin* sau *cout* pentru a efectua operații de intrare și ieșire. Lecția de față v-a prezentat o serie de modificatori și de funcții pentru operațiile I/E cu fluxurile *cin* și *cout*. Pe măsură ce devin mai complexe, programele vor plasa adesea informațiile în fișiere. În lecția 36, „Operații de intrare/ieșire cu fișiere în C++", veți învăța să efectuați în C++ operații de intrare și ieșire cu fișiere. Dar, înainte de a trece la lecția 36, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Fișierul antet *iostream.h* definește fluxurile *cin* și *cout* ca variabile obiect aparținând claselor *istream* și, respectiv, *ostream*. Ca și obiecte, *cin* și *cout* oferă metode pe care programele le pot apela pentru efectuarea de operații specifice.

☑ Metoda *cout.width* permite programelor să specifice numărul minim de caractere cu care va fi afișată valoarea următoare.

☑ Metoda *cout.fill* permite programelor să precizeze caracterul care va fi utilizat de *cout* la umplerea spațiilor pentru *cout.width* sau *setw*.

☑ Modificatorul *setprecision* permite programelor să controleze numărul de cifre afișate după virgulă în cazul valorilor în virgulă mobilă.

☑ Metodele *cin.get* și *cout.put* permit programelor să citească sau să afișeze un singur caracter odată.

☑ Metoda *cin.getline* permite programelor să citească de la tastatură o întreagă linie de text.

Lecția 36

Operații de intrare/ieșire cu fișiere în C++

Pe măsură ce cresc în complexitate, programele ajung să scrie și să citească informații cu ajutorul fișierelor. Dacă operațiile cu fișiere vă sunt deja familiare din limbajul C, metode similare se pot folosi și în C++. În plus, așa cum veți vedea în lecția de față, C++ oferă o serie de clase de fluxuri fișier care fac ca operațiile de intrare și ieșire (I/E) cu fișiere să fie foarte simple. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Prin intermediul unui flux fișier de ieșire, puteți să scrieți informații într-un fișier folosind *operatorul de inserare* (<<).
- Prin intermediul unui flux fișier de intrare, puteți să citiți informațiile aflate într-un fișier folosind *operatorul de extragere* (>>).
- Deschiderea și închiderea fișierelor se fac prin intermediul metodelor de clasă.
- Pentru citirea și scrierea de date în fișiere puteți folosi operatorii de inserare și de extragere sau puteți apela la o serie de metode de clasă.

Multe dintre programele pe care le veți crea în viitor vor utiliza intens fișiere. Luați-vă timpul necesar pentru a experimenta cu programele prezentate în această lecție. Așa cum veți vedea, C++ face ca operațiile cu fișiere să fie foarte ușoare.

Scrierea de date într-un flux fișier

În lecția 35, „Alte operații cu cin și cout", ați văzut că *cout* este un obiect de tip *ostream* (flux de ieșire). Cu ajutorul clasei *ostream*, programele pot efectua operații de ieșire cu *cout* folosind operatorul de inserare sau diferite metode de clasă, precum *cout.put*. Fluxul de ieșire *cout* este definit în fișierul de antet *iostream.h*.

Într-un mod similar, fișierul de antet *fstream.h* definește o clasă ce reprezintă un flux fișier de ieșire numit *ofstream*. Prin intermediul obiectelor *ofstream*, programele pot efectua operații de ieșire cu fișiere. Pentru început, trebuie să declarați un obiect *ofstream*, specificând numele de fișier ca șir de caractere, așa cum se vede aici:

```
ofstream obiect_fisier("NUME_FIS.EXT");
```

Atunci când specificați un nume de fișier în declarația unui obiect *ofstream*, C++ creează pe disc un nou fișier care va avea numele precizat. Dacă există deja un fișier cu acel nume, C++ va suprascrie conținutul fișierului. Programul următor, *Out_File.CPP*, creează un obiect *ofstream* și apoi folosește operatorul de inserare pentru a scrie în fișierul *BookInfo.DAT* mai multe linii de text:

```
#include <fstream.h>

void main(void)
  {
    ofstream book_file("BookInfo.DAT");

    book_file << "Rescued by C++, Third Edition" << endl;
    book_file << "Jamsa Press" << endl;
    book_file << "29.95" << endl;
  }
```

În exemplul anterior, programul deschide fișierul *BookInfo.DAT* și apoi scrie în el trei linii, folosind operatorul de inserare. Compilați și rulați programul *Out_File.CPP*. Dacă lucrați în mediul MS-DOS, puteți utiliza comanda *TYPE* pentru a afișa conținutul fișierului, așa cum se ilustrează mai jos:

```
C:\> TYPE BookInfo.DAT <Enter>
Rescued by C++, Third Edition
Jamsa Press
$29.95
```

Precum vedeți, C++ face ca operațiile elementare de scriere în fișier să fie foarte simple.

Citirea dintr-un flux fișier de intrare

După cum tocmai ați învățat, programele pot efectua ușor operații de scriere în fișiere cu ajutorul clasei *ofstream*. Într-un mod similar, programele pot efectua operații de citire din fișiere prin intermediul obiectelor *ifstream*. Pentru a accesa un fișier în scopul operațiilor de intrare, este suficient să creați un obiect și să transmiteți numele fișierului dorit către metoda constructor a acestui obiect, ca mai jos:

```
ifstream fisier_citire("NUME_FIS.EXT");
```

Programul următor, *File_In.CPP*, deschide fișierul *BookInfo.DAT* pe care l-ați creat cu programul anterior, citind și afișând primele trei intrări din fișier:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
    ifstream input_file("BookInfo.DAT");

    char one[64], two[64], three[64];
```

```
    input_file >> one;
    input_file >> two;
    input_file >> three;
    cout << one << endl;
    cout << two << endl;
    cout << three << endl;
  }
```

După compilarea şi rularea programului *File_In.CPP*, v-aţi putea aştepta ca programul să afişeze primele trei linii din fişier. Asemenea lui *cin*, însă, fluxul fişier de intrare foloseşte caractere de spaţiere pentru a determina sfârşitul unei valori şi începutul celei următoare. În consecinţă, atunci când executaţi programul de mai sus, pe ecran vor fi afişate următoarele:

```
C:\> File_In <Enter>
Rescued
by
C++,
```

Citirea din fişier a unei întregi linii

În lecţia 35, „Alte operaţii cu cin şi cout", aţi văzut că programele pot folosi *cin.getline* pentru a citi de la tastatură o întreagă linie. În mod similar, obiectele *ifstream* pot utiliza *getline* pentru a citi dintr-un fişier o linie completă. Următorul program, *FileLine.CPP*, apelează la metoda *getline* pentru a citi toate cele trei linii ale fişierului *BookInfo.DAT*:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
  {
    ifstream input_file("BookInfo.DAT");
    char one[64], two[64], three[64];
    input_file.getline(one, sizeof(one))
    input_file.getline(two, sizeof(two));
    input_file.getline(three, sizeof(three));
    cout << one << endl;
```

```
        cout << two << endl;
        cout << three << endl;
    }
```

În acest exemplu, programul reușește să citească întreg conținutul fișierului deoarece știe că respectivul fișier conține trei linii. De multe ori, însă, programul nu va cunoaște câte linii se află într-un fișier. În asemenea situații, programul va continua să citească din fișier până la întâlnirea sfârșitului de fișier, așa cum vom discuta în cele ce urmează.

Detectarea sfârșitului de fișier

Una dintre operațiile cu fișiere uzuale într-un program este citirea conținutului unui fișier până la sfârșitul acestuia. Pentru a detecta sfârșitul de fișier, programele pot apela la metoda *eof* oferită de obiect. Această metodă întoarce valoarea 0 în cazul în care sfârșitul fișierului nu a fost atins încă și 1 în caz contrar. Prin intermediul unei instrucțiuni *while*, programele pot continua citirea datelor din fișier până la sfârșitul acestuia, așa cum este ilustrat aici:

```
while (! fisier_citire.eof())
  {
    // Instructiuni
  }
```

În acest exemplu, programul va continua să cicleze atât timp cât metoda *eof* întoarce fals (0). Programul următor, *Test_Eof.CPP*, folosește metoda *eof* pentru a citi conținutul fișierului *BookInfo.DAT* până la întâlnirea sfârșitului de fișier:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
    ifstream input_file("BookInfo.DAT");

    char line[64];

    while (! input_file.eof())
    {
      input_file.getline(line, sizeof(line));

      cout << line << endl;
    }
  }
```

Similar cu *Test_Eof.CPP*, programul următor, *Word_EOF.CPP*, citeşte conţinutul fişierului cuvânt cu cuvânt, până când detectează sfârşitul de fişier:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
    ifstream input_file("BookInfo.DAT");

    char word[64];

    while (! input_file.eof())
    {
      input_file >> word;

      cout << word << endl;
    }
  }
```

După cum puteţi vedea, programul *Word_EOF.CPP* foloseşte operatorul de extragere (>>) pentru a citi câte un cuvânt din fluxul fişier de intrare.

În fine, programul de mai jos, *Char_EOF.CPP*, citeşte conţinutul fişierului caracter cu caracter, prin intermediul metodei *get*, până la întâlnirea sfârşitului de fişier:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
    ifstream input_file("BookInfo.DAT");

    char letter;

    while (! input_file.eof())
    {
      letter = input_file.get();

      cout << letter;
    }
  }
```

Detectarea erorilor în operațiile cu fișiere

Până în acest moment, programele din lecția de față au presupus că în timpul operațiilor I/E cu fișiere nu apare nici o eroare. Din păcate, lucrurile nu stau așa întotdeauna. Spre exemplu, atunci când deschide un fișier pentru citire, programul ar trebui să se asigure că respectivul fișier există. De asemenea, atunci când scrie date într-un fișier, programul ar trebui să se asigure că operația de scriere a reușit (o eroare de lipsă de spațiu pe disc, de pildă, poate împiedica scrierea datelor în fișierul aflat pe disc). Pentru detectarea erorilor în cadrul programelor puteți apela la metoda *fail* a unui obiect fișier. În cazul în care o operație decurge fără erori, metoda *fail* întoarce valoarea fals (0). Dacă apare o eroare, însă, metoda *fail* va întoarce valoarea adevărat. De exemplu, atunci când programul deschide un fișier, acesta ar trebui să folosească metoda *fail* pentru a determina dacă s-a produs o eroare, ca mai jos:

```
ifstream input_file("FILENAME.DAT");

if (input_file.fail())
  {
    cerr << "Error opening FILENAME.EXT" << endl;
    exit(1);
  }
```

Într-un mod similar, programul ar trebui să verifice și succesul operațiilor de citire și scriere în fișier. Următorul program, *Test_All.CPP*, utilizează metoda *fail* pentru a testa diferite condiții de eroare:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
    char line[256];

    ifstream input_file("BookInfo.DAT");

    if (input_file.fail())
      cerr << "Error opening BookInfo.DAT" << endl;
    else
      {
        while ((! input_file.eof()) &&
          (! input_file.fail()))
          {
```

```
                  input_file.getline(line, sizeof(line));
                  if (! input_file.fail())
                    cout << line << endl;
            }
      }
}
```

Închiderea unui fișier după utilizare

La încheierea unui program, sistemul de operare închide toate fișierele pe care acel program le-a deschis. Ca o regulă, însă, este bine ca programele să închidă un fișier imediat ce acesta nu mai este folosit. Pentru închiderea unui fișier, programul va folosi metoda *close*, ca aici:

```
fisier_citire.close();
```

Închiderea unui fișier face ca sistemul de operare să depună (să scrie) pe disc toate datele pe care programul le-a scris în fișier și să actualizeze intrarea din director a fișierului respectiv.

Controlarea modului de deschidere a unui fișier

În fiecare din programele prezentate ca exemple în această lecție, operațiile de citire și scriere în fișiere au pornit de la începutul fișierului. Atunci când un program scrie date într-un fișier de ieșire, însă, apar situații în care doriți ca programul să anexeze informațiile la sfârșitul fișierului. Pentru a deschide un fișier în *modul anexare*, trebuie să specificați la deschidere un al doilea parametru (*ios::app*), ca mai jos:

```
ifstream fisier_scriere("NUME_FIS.EXT", ios::app);
```

În acest caz, parametrul *ios::app* indică un *mod de deschidere a fișierului*. Odată ce devin mai complexe, programele utilizează o combinație între valorile pentru moduri de deschidere pe care le prezintă tabelul 36.

Mod de deschidere	Scop
ios::app	Deschide un fișier în modul anexare, plasând indicatorul de poziție la sfârșitul fișierului.
ios::ate	Plasează indicatorul de poziție la sfârșitul fișierului.
ios::in	Deschide fișierul pentru citire.
ios::nocreate	Dacă fișierul nu există, nu creează un nou fișier și întoarce o eroare.

Tabelul 36 *Valori pentru modurile de deschidere a fișierelor*

Mod de deschidere	Scop
ios::noreplace	Dacă fișierul există, operația de deschidere eșuează și întoarce o eroare.
ios::out	Deschide fișierul pentru scriere.
ios::trunc	Trunchiază (suprascrie) conținutul unui fișier existent.

***Tabelul 36** Valori pentru modurile de deschidere a fișierelor (continuare)*

După cum ați aflat, atunci când un program deschide un fișier pentru scriere, metodele pentru fișiere suprascriu, în mod normal, conținutul unui fișier existent. În funcție de scopul programului, pot exista situații în care să nu doriți suprascrierea unui fișier existent. Următoarea instrucțiune, de pildă, deschide un fișier pentru scriere, folosind modul *ios::noreplace* pentru a împiedica programul să suprascrie un fișier existent:

```
ifstream fisier_scriere("NUME_FIS.EXT", ios::out |
  ios::noreplace);
```

Efectuarea operațiilor de citire și de scriere

Programele de până acum, din această lecție, au efectuat operații cu fișiere utilizând exclusiv șiruri de caractere. Pe măsură ce programele cresc în complexitate, pot apărea cazuri în care să aveți nevoie de citirea și scrierea de vectori și structuri. În astfel de situații, programele pot apela la metodele *read* și *write*. Utilizarea metodei *read* se face prin specificarea unei zone tampon în care metoda va depune datele citite și a unui al doilea parametru ce indică dimensiunea în octeți a zonei, ca mai jos:

```
fisier_citire.read(tampon, sizeof(tampon));
```

De asemenea, utilizarea metodei *write* presupune specificarea zonei tampon care conține datele pe care metoda le va scrie în fișier și a unui parametru care indică dimensiunea zonei, ca mai jos:

```
fisier_scriere.write(tampon, sizeof(tampon));
```

Spre exemplu, programul următor, *Stru_Out.CPP*, utilizează metoda *write* pentru a scrie conținutul unei structuri în fișierul *Employee.DAT*:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
  struct employee {
    char name[64];
```

```
        int age;
        float salary;
    } worker = { "John Doe", 33, 25000.0 };

    ofstream emp_file("Employee.DAT");

    emp_file.write((char *) &worker, sizeof(employee));
    }
```

Metoda *write* așteaptă, în mod normal, un pointer la un șir de caractere. În cadrul apelului *write*, literele *(char *)* reprezintă o *conversie de tip*, care informează compilatorul că pointerul transmis are un tip diferit. Dacă omiteți această conversie, compilatorul de C++ va genera o eroare de sintaxă deoarece funcția așteaptă un pointer la tipul *char*. În mod similar, programul următor, *Stru_In.CPP*, folosește metoda *read* pentru a citi din fișier informațiile despre un angajat:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
  struct employee {
    char name[64];
    int age;
    float salary;
  } worker = { "John Doe", 33, 25000.0 };

  ifstream emp_file("Employee.DAT");

  emp_file.read((char *) &worker, sizeof(employee));

  cout << worker.name << endl;
  cout << worker.age << endl;
  cout <<worker.salary << endl;
  }
```

Ce trebuie să știți

Pe măsură ce programele dumneavoastră vor spori în complexitate, operațiile cu fișiere vor deveni uzuale. Experimentați, o vreme, cu programele prezentate în această lecție. În lecția 37, „Funcții inline și cod în limbaj de asamblare", veți învăța să îmbunătățiți performanțele programelor cu ajutorul funcțiilor inline. Dar, înainte de a trece la lecția 37, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Fișierul de antet *fstream.h* definește clasele *ifstream* și *ofstream*, cu ajutorul cărora programele pot să efectueze operații de intrare și ieșire cu fișiere.
- ☑ Pentru a deschide un fișier în scopul operațiilor de citire sau scriere, programele declară un obiect *ifstream* sau *ofstream*, transmițând numele fișierului către metoda constructor a obiectului.
- ☑ După deschiderea unui fișier pentru citire sau scriere, programele pot citi sau scrie date prin intermediul operatorilor de extragere (>>) și de inserare (<<).
- ☑ Programele pot citi și scrie în fișiere câte un singur caracter prin intermediul metodelor *get* și *put*.
- ☑ Programele pot citi dintr-un fișier o întreagă linie prin intermediul metodei *getline*.
- ☑ Cele mai multe programe citesc conținutul unui fișier până la sfârșitul acestuia. Detectarea în program a sfârșitului de fișier se poate face cu ajutorul metodei *eof*.
- ☑ Atunci când efectuează operații cu fișiere, programele ar trebui să testeze rezultatele diverselor operații pentru a se asigura de îndeplinirea cu succes a acestora. Detectarea erorilor în programe se poate face cu ajutorul metodei *fail*.
- ☑ În cazul în care este necesară citirea sau scrierea de date precum o structură sau un vector, programele pot apela la metodele *read* și *write*.
- ☑ După utilizarea unui fișier, programele ar trebui să închidă acel fișier cu ajutorul metodei *close*.

Lecția 37
Funcții inline și cod în limbaj de asamblare

Începând cu lecția 11, „O introducere în funcții", programele și clasele create au folosit intensiv funcții. După cum ați învățat, singurul dezavantaj al utilizării funcțiilor este efortul suplimentar (creșterea duratei operațiilor efectuate de program) implicat de plasarea parametrilor pe stivă pentru fiecare apel. Așa cum veți vedea în lecția de față, în cazul funcțiilor scurte puteți recurge la ceea ce se numește *cod inline*, adică plasarea instrucțiunilor unei funcții în locul fiecărui apel din program al acelei funcții – ceea ce elimină efortul de apelare a funcțiilor. Prin intermediul funcțiilor inline, programele vor rula ceva mai repede. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Pentru creșterea performanțelor prin reducerea efortului de apelare a funcțiilor, puteți determina compilatorul C++ să plaseze inline codul unei funcții, destul de similar cu înlocuirea cu macrodefiniții.
- Prin intermediul *funcțiilor inline*, programele își conservă lizibilitatea (oricine citește programul vede apelul de funcție), dar se elimină efortul de apelare a funcțiilor datorat depunerii și extragerii de valori din stivă. Se elimină, de asemenea, și efortul pe care îl presupune saltul către și de la funcții.
- În funcție de necesitățile programului, pot exista situații în care să fie necesară utilizarea limbajului de asamblare pentru îndeplinirea unei anumite sarcini.
- Pentru a simplifica utilizarea secvențelor de cod în limbaj de asamblare, C++ vă permite specificarea în programe a funcțiilor inline în limbaj de asamblare.

Despre funcțiile inline

Atunci când definiți o funcție în program, compilatorul de C++ convertește acea funcție în limbaj de asamblare, reținând în program doar o copie a instrucțiunilor respective. De fiecare dată când programul apelează funcția, compilatorul de C++ plasează în program instrucțiuni speciale, care depun parametrii pe stivă și apoi ramifică execuția programului la instrucțiunile funcției. După încheierea funcției, execuția programului continuă cu prima instrucțiune ce urmează apelului de funcție. Codul care plasează parametrii pe stivă și care determină ca execuția programului să se ramifice și apoi să revină din setul de instrucțiuni ale funcției, implică un efort ce face ca programul să fie ceva mai încet decât în cazul în care aceleași instrucțiuni ar fi fost plasate în locul fiecărui apel de funcție. Să luăm spre exemplu programul următor, *CallBeep.CPP*, care apelează funcția *show_message*, aceasta producând un anumit număr de sunete în difuzorul calculatorului și afișând apoi un mesaj:

```
#include <iostream.h>

void show_message(int count, char *message)
  {
    int i;
    for (i = 0; i < count; i++)
      cout << '\a'
    cout << message << endl;
  }

void main(void)
  {
    show_message(3, "Rescued by C++");
    show_message(2, "Lesson 37")
  }
```

Programul următor, *No_Call.CPP*, nu mai apelează funcția *show_message*. În schimb, instrucțiunile funcției sunt plasate în locul fiecărui apel de funcție, așa cum se poate vedea:

```
#include <iostream.h>

void main(void)
  {
    int i;
    for (i = 0; i < 3; i++)
      cout << '\a'
    cout << "Rescued by C++" << endl;
    for (i = 0; i < 2; i++)
      cout << '\a'
    cout << "Lesson 37" << endl;
  }
```

Ambele programe efectuează aceleași operații. Deoarece nu apelează funcția *show_message*, programul *No_Call* rulează ceva mai repede decât programul *CallBeep*. În acest caz, diferența dintre timpii de execuție este imposibil de detectat; dacă programul ar fi apelat funcția de 1000 de ori, însă, o ușoară creștere a performanței ar putea deveni sesizabilă. Pe de altă parte, programul *No_Call* este mai aglomerat decât perechea sa, *CallBeep*, și poate fi, din această cauză, mai dificil de înțeles.

Atunci când creați programe, încercați întotdeauna să determinați când este mai avantajoasă utilizarea funcțiilor și când este de preferat să recurgeți la funcții inline. Pentru majoritatea programelor simple, utilizarea funcțiilor este, de regulă, alegerea cea mai bună. Dacă scrieți, însă, un program pentru care performanțele sunt cruciale, ați putea fi nevoit să reduceți numărul de apeluri de funcții efectuate în program.

O cale de reducere a apelurilor de funcții este să plasați instrucțiunile corespunzătoare pe toată întinderea programului, așa cum se întâmplă în programul *No_Call*. După cum ați văzut, însă, eliminarea unei singure funcții a dus la o aglomerare considerabilă a programului. Din fericire, C++ oferă cuvântul cheie *inline* prin intermediul căruia puteți profita de avantajele ambelor alternative.

Utilizarea cuvântului cheie inline

La declararea unei funcții în program, C++ vă permite precedarea numelui de funcție cu cuvântul cheie *inline*. Atunci când întâlnește cuvântul cheie *inline*, compilatorul de C++ va plasa instrucțiunile funcției în locul fiecărui apel de funcție în cadrul fișierului executabil (în cod mașină). În acest fel, programul C++ își sporește lizibilitatea prin utilizarea funcțiilor, îmbunătățindu-și totodată performanțele prin eliminarea efortului de apelare a funcțiilor. Programul următor, *Inline.CPP*, definește funcțiile *max* și *min* ca *inline*:

```
#include <iostream.h>

inline int max(int a, int b)
  {
    if (a > b)
      return(a);
    else
      return(b);
  }

inline int min(int a, int b)
  {
    if (a < b)
      return(a);
    else
      return(b);
  }

void main(void)
  {
    cout << "The min of 1001 and 2002 is " <<
      min(1001, 2002) << endl;
```

```
    cout << "The max of 1001 and 2002 is " <<
      max(1001, 2002) << endl;
}
```

În acest exemplu, compilatorul de C++ va înlocui fiecare apel de funcție cu instrucțiunile corespunzătoare. Performanța programului crește, astfel, fără ca programul să devină mai dificil de înțeles.

Despre funcțiile inline



Atunci când întâlnește cuvântul cheie *inline* înaintea unei definiții de funcții, compilatorul de C++ va înlocui mai apoi fiecare referință (apelurile) la acea funcție cu instrucțiunile pe care le conține funcția. În acest fel, performanțele la rularea programului cresc prin eliminarea consumului implicat de apelurile de funcții, în timp ce sporul de lizibilitate adus de utilizarea funcției rămâne neafectat.

Funcții inline și clase

Așa cum ați văzut, la definirea unei clase, funcțiile acesteia pot fi definite în interiorul sau în exteriorul clasei. Spre exemplu, clasa *employee* de mai jos își definește funcțiile în interiorul său:

```
class employee {
  public:
    employee(char *name, char *position, float salary)
    {
      strcpy(employee::name, name);
      strcpy(employee::position, position);
      employee::salary = salary;
    }
    void show_employee(void)
    {
      cout << "Name: " << name << endl;
      cout << "Position: " << position << endl;
      cout << "Salary: $" << salary << endl;
    }
  private:
```

```
        char name[64];
        char position[64];
        float salary;
};
```

Programatorii denumesc adesea procesul de plasare a funcţiilor în interiorul unei clase ca plasare inline a funcţiilor. Atunci când creaţi astfel de funcţii inline în clase, C++ multiplică funcţiile pentru fiecare obiect creat, plasând codul inline la fiecare apel al metodei. Avantajul acestui cod inline este sporul de performanţă. Dezavantajul este faptul că programele pot creşte repede în dimensiuni, în funcţie de maniera de utilizare a obiectelor. În plus, plasarea codului de funcţii în interiorul unei definiţii de clasă poate aglomera acea definiţie, făcând ca membrii prezenţi să fie dificil de înţeles atât pentru dumneavoastră, cât şi pentru alţi programatori.

Pentru a spori lizibilitatea definiţiilor de clase, puteţi să extrageţi funcţiile unei clase din interiorul definiţiei acesteia şi apoi să le precedaţi cu cuvântul cheie *inline*. De exemplu, următoarea definiţie de funcţie determină compilatorul să utilizeze instrucţiuni inline în cazul funcţiei *show_employee*:

```
inline void employee:: show_employee(void)
  {
    cout << "Name: " << name << endl;
    cout << "Position: " << position << endl;
    cout << "Salary: $" << salary << endl;
  }
```

Utilizarea instrucţiunilor în limbaj de asamblare

Aşa cum aţi aflat din lecţia a doua, „Crearea primului program", programatorii pot scrie aplicaţii într-o varietate de limbaje de programare. Apoi, prin intermediul unui compilator, programatorii transformă instrucţiunile de program în codul maşină (unu şi zero) pe care calculatorul îl înţelege. Fiecare tip de calculator acceptă un limbaj intermediar, numit *limbaj de asamblare*, care se situează undeva între limbajul maşină şi limbajele de programare, precum C++.

Limbajul de asamblare foloseşte propriile simboluri pentru a reprezenta instrucţiunile în cod maşină. În funcţie de scopurile programelor, pot exista cazuri în care un program avansat trebuie să efectueze operaţii la un nivel de bază, operaţii care presupun utilizarea instrucţiunilor în limbaj de asamblare. În asemenea situaţii, aţi putea folosi instrucţiunea *asm* din C++ pentru a insera în program instrucţiuni în limbaj de asamblare. Majoritatea programelor pe care le veţi scrie nu vor necesita cod în limbaj de asamblare. Programul următor, *Use_Asm.CPP*, utilizează instrucţiunea *asm* pentru a insera codul în limbaj de asamblare care generează un sunet în difuzorul calculatorului, sub mediul MS-DOS:

```
#include <iostream.h>

void main(void)
  {
    cout << "About to sound the bell!" << endl;
    asm {
      MOV AH,2
      MOV DL,7
      INT 21H
    }
    cout << "Done!" << endl;
  }
```

După cum puteți vedea, folosind instrucțiunea *asm*, programul combină instrucțiunile C++ cu cele în limbaj de asamblare.

Ce trebuie să știți

Funcțiile inline cresc performanțele programelor prin reducerea efortului implicat de apelurile de funcții. În lecția de față ați învățat cum și când să folosiți într-un program funcții inline. Ați văzut, de asemenea, că pot exista cazuri în care programele trebuie să utilizeze limbajul de asamblare pentru a îndeplini anumite sarcini. În lecția 38, „Utilizarea parametrilor din linia de comandă", veți vedea cum pot programele să acceseze parametrii din linia de comandă, pe care utilizatorul îi precizează la rularea programului. Dar, înainte de a trece la lecția 38, asigurați-vă că ați reținut următoarele aspecte importante:

☑ Plasarea parametrilor pe stivă și saltul către și de la instrucțiunile funcției implică un efort care are ca efect o rulare mai înceată a programului.

☑ Cuvântul cheie *inline* determină compilatorul de C++ să înlocuiască un apel de funcție cu instrucțiuni echivalente celor conținute în funcția respectivă. Deoarece instrucțiunile plasate inline elimină efortul de apelare a funcției, programul va rula mai rapid.

☑ Atunci când folosiți funcții inline în cadrul unei clase, fiecare obiect creat utilizează propriile instrucțiuni inline. În mod normal, obiectele aceleiași clase partajează același cod de funcții.

☑ Cuvântul cheie *asm* permite inserarea în programele C++ a instrucțiunilor în limbaj de asamblare.

Lecția 38

Utilizarea parametrilor din linia de comandă

Așa cum știți, atunci când executați comenzi în cadrul promptului de sistem, de cele mai multe ori aveți posibilitatea să precizați în linia de comandă informații adiționale, precum un nume de fișier. Spre exemplu, la utilizarea comenzii *COPY* din MS-DOS în scopul copierii conținutului unui fișier într-un altul, în linia de comandă se specifică numele ambelor fișiere. De asemenea, în cazul în care compilatorul funcționează în linie de comandă, la apelarea acestuia trebuie precizat numele fișierului sursă. Lecția de față studiază modul în care programele C++ accesează parametrii din linia de comandă. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Programele C++ consideră parametrii din linia de comandă ca parametri pentru funcția *main*.
- În mod exact, C++ transmite doi (uneori trei) parametri funcției *main*, numiți în cele mai multe programe *argc* și *argv*.
- Parametrul *argc* conține numărul de parametri din linia de comandă pe care sistemul de operare îi transmite programului.
- Parametrul *argv* este un vector de pointeri la șiruri de caractere, fiecare șir corespunzând unui parametru din linia de comandă.
- În funcție de compilator, este posibil ca programul să aibă acces la un al treilea parametru, *env*, care este un vector de pointeri la șiruri de caractere corespunzând variabilelor de mediu.

Capacitatea unui program de a accesa parametrii din linia de comandă are ca efect creșterea posibilităților de utilizare a unui aceluiași program. De exemplu, puteți crea propriul program de copiere pe care să-l utilizați pentru a copia orice fișier sursă, specificat ca un prim parametru din linia de comandă, în orice fișier destinație, specificat ca un al doilea parametru din linia de comandă. Deoarece programul de copiere acceptă parametri în linia de comandă, același program poate fi utilizat pentru a copia orice număr de fișiere.

Utilizarea parametrilor argv și argc

Atunci când rulați un program de la promptul sistemului, comanda tastată reprezintă linia de comandă a programului, ca mai jos:

```
C:\> COPY Sursa.DOC Destinat.DOC <Enter>
```

În acest exemplu, linia de comandă conține comanda propriu-zisă (*COPY*) și doi parametri (numele de fișiere *Sursa.DOC* și *Destinat.DOC*). Pentru a permite programelor să acceseze linia de comandă, C++ transmite funcției *main* doi parametri:

```
void main(int argc, char *argv[])
```

Primul parametru, *argc*, conține numărul de elemente din vectorul *argv*. Luând ca exemplu comanda *COPY* de mai sus, parametrul *argc* ar avea valoarea 3 (numele comenzii și cei doi parametri). Programul următor, *ShowArgc.CPP*, folosește parametrul *argc* pentru a afișa numărul parametrilor din linia de comandă:

```
#include <iostream.h>

void main(int argc, char *argv[])
  {
    cout << "Number of command-line arguments is " <<
      argc << endl;
  }
```

Experimentați puțin cu programul *ShowArgc.CPP*, apelându-l cu numere diferite de parametri, ca aici:

```
C:\> ShowArgc A B C <Enter>
Number of command-line arguments is 4
```

În funcție de tipul său, compilatorul ar putea considera elementele grupate prin ghilimele ca fiind un singur parametru, așa cum se vede aici:

```
C:\> ShowArgc "Acesta este un singur parametru" <Enter>
Number of command-line arguments is 2
```

Al doilea parametru al funcției *main*, *argv*, este un vector de pointeri la șirurile de caractere ce conțin fiecare intrare din linia de comandă. Figura 38.1 ilustrează, de exemplu, modul în care elementele vectorului *argv* indică intrările din linia de comandă.

Următorul program, *ShowArgv.CPP*, utilizează o instrucțiune *for* pentru a afișa elementele vectorului *argv* (linia de comandă a programului). Programul începe cu primul element din vector (numele programului) și apoi afișează fiecare element până când valoarea variabilei de control a buclei depășește valoarea *argc* (care reprezintă numărul de intrări din linia de comandă):

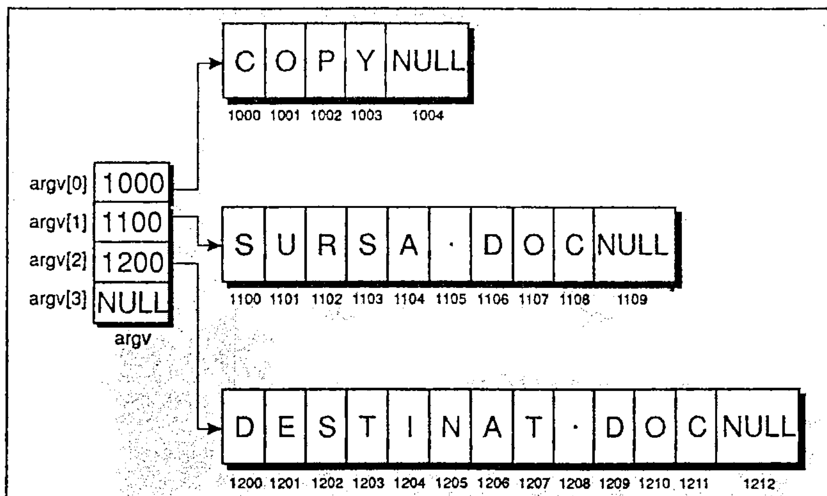


Figura 38.1 *Vectorul argv indică parametrii din linia de comandă.*

```
#include <iostream.h>

void main(int argc, char *argv[])
  {
    int i;

    for (i = 0; i < argc; i++)
      cout << "argv[" << i << "] contains " << argv[i] <<
        endl;
  }
```

După compilarea programului *ShowArgv.CPP*, rulați acest program printr-o linie de comandă similară cu următoarea:

```
C:\> ShowArgv A B C <Enter>
argv[0] contains ShowArgv.EXE
argv[1] contains A
argv[2] contains B
argv[3] contains C
```

Accesarea parametrilor din linia de comandă



Pentru a spori numărul de operații pe care le poate efectua un program, C++ permite programelor să acceseze parametrii din linia de comandă cu ajutorul a doi parametri care sunt transmiși funcției *main*. Primul parametru, *argc*, conține numărul parametrilor din linia de comandă (inclusiv numele programului). Cel de al doilea parametru, *argv*, este un vector de pointeri la șiruri de caractere. Fiecare dintre aceste șiruri de caractere corespunde câte unui parametru din linia de comandă. Pentru a accesa parametrii din linia de comandă trebuie să modificați antetul funcției *main* în felul următor:

```
void main(int argc, char *argv[])
```

Repetarea buclei până când argv este NULL

Așa cum ați învățat, programele C++ folosesc caracterul *NULL* pentru a încheia șirurile de caractere. Într-un mod similar, C++ folosește un caracter *NULL* pentru a marca ultimul element din vectorul *argv*. Programul următor, *ArgvNULL.CPP*, modifică instrucțiunea *for* din programul *ShowArgv.CPP* pentru a parcurge elementele vectorului *argv* până când elementul curent devine *NULL*:

```
#include <iostream.h>

void main(int argc, char *argv[])
  {
    int i;

    for (i = 0; argv[i] != NULL; i++)
      cout << "argv[" << i << "] contains   << argv[i] <<
     endl;
  }
```

Tratarea parametrului argv ca pointer

După cum ați învățat, C++ vă permite să accesați vectori cu ajutorul pointerilor. Programul următor, *ArgvPtr.CPP*, tratează *argv* ca un pointer la un pointer la șir de caractere (cu alte cuvinte, un pointer la pointer) pentru a afișa conținutul liniei de comandă:

```
#include <iostream.h>

void main(int argc, char **argv)
  {
```

```
    int i = 0;

    while (*argv)
       cout << "argv[" << i++ << "] contains " << *argv++ <<
         endl;
  }
```

Priviți cu atenție declarația parametrului *argv* din *main*:

```
void main(int argc, char **argv)
```

Primul asterisc din declarație arată compilatorului de C++ că *argv* este un pointer. Cel de al doilea asterisc spune compilatorului că *argv* este un pointer la un alt pointer – în cazul nostru, un pointer la un pointer de tip *char*. Gândiți-vă la *argv* ca la un vector de pointeri. Fiecare element al vectorului indică, în exemplul nostru, un vector de tip *char*.

Utilizarea parametrilor din linia de comandă

Următorul program, *FileShow.CPP*, folosește parametrii din linia de comandă pentru a afișa pe ecran conținutul unui fișier specificat de utilizator. Pentru a folosi programul *FileShow* în scopul afișării pe ecran a conținutului fișierului *AUTOEXEC.BAT* din directorul rădăcină, de exemplu, linia de comandă se prezintă astfel:

```
C:\> FileShow \AUTOEXEC.BAT <Enter>
```

Instrucțiunile următoare reprezintă implementarea programului *FileShow.CPP*. Programul începe prin verificarea parametrului *argc*, pentru a se asigura că utilizatorul a specificat un fișier în linia de comandă. Dacă utilizatorul a precizat numele de fișier, parametrul *argc* va conține valoarea 2. În continuare, programul deschide fișierul și îi afișează conținutul, folosind tehnicile despre care ați învățat în lecția 36, „Operații de intrare/ieșire cu fișiere în C++". După cum puteți vedea, în cazul în care nu reușește să deschidă fișierul specificat de utilizator, programul afișează un mesaj de eroare și se încheie:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char *argv[])
  {
    char line[256];

    if (argc < 2)
    {
```

```
      cerr << "You must specify a filename" << endl;
      exit(1);
   }
   ifstream input_file(argv[1]);
   if (input_file.fail())
   cerr << "Error opening BookInfo.DAT" << endl;
   else
   {
      while ((! input_file.eof()) && (! input_file.fail()))
         {
            input_file.getline(line, sizeof(line));
            if (! input_file.fail())
               cout << line << endl;
         }
   }
}
```

Accesarea variabilelor de mediu din sistemul de operare

Precum știți, majoritatea sistemelor de operare vă permit definirea unor variabile de mediu pe care programele le pot accesa pentru determinarea unor diverși parametri, așa cum este calea de comandă. Dacă utilizați, de pildă, mediul MS-DOS, stabilirea variabilelor de mediu se face prin intermediul comenzii *SET*. În funcție de tipul compilatorului, este posibil să puteți accesa din program aceste variabile de mediu cu ajutorul unui al treilea parametru al funcției *main*, numit *env*. Asemeni parametrului *argv*, *env* este vector de pointeri la șiruri de caractere. Și tot asemeni lui *argv*, C++ încheie acest vector cu un caracter *NULL*. În cazul în care compilatorul pe care-l folosiți acceptă parametrul *env*, puteți modifica antetul funcției *main* în felul următor:

```
void main(int argc, char *argv[], char *env[])
```

Programul următor, *ShowEnv.CPP*, parcurge iterativ elementele vectorului *env* pentru a afișa valorile variabilelor de mediu:

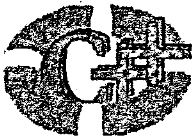
```
#include <iostream.h>
void main(int argc, char *argv[], char *env[])
   {
     while (*env)
```

```
        cout << *env++ << endl;
    }
```

Așa cum puteți observa, programul parcurge iterativ elementele vectorului *env* până când întâlnește pointerul *NULL*, care indică ultima intrare din vector. După compilarea și rularea programului *ShowEnv.CPP*, programul va afișa valorile variabilelor de mediu, ca mai jos:

```
C:\> ShowEnv <Enter>
TEMP=C:\WINDOWS\TEMP
PROMPT=$p$g
COMSPEC=C:\WINDOWS\COMMAND.COM
PATH=C:\WINDOWS;C:\DOS
```

Accesarea variabilelor de mediu



În funcție de tipul compilatorului, este posibil ca programele să poată accesa variabilele de mediu ale sistemului de operare prin intermediul unui al treilea parametru al funcției *main*, *env*. Asemeni parametrului *argv*, *env* este un vector de pointeri la șiruri de caractere, fiecare dintre acestea indicând câte o variabilă de mediu. Pentru a putea accesa variabilele de mediu cu ajutorul parametrului *env*, modificați antetul funcției *main* astfel:

```
void main(int argc, char *argv[], char *env[])
```

Ce trebuie să știți

Pentru a crește numărul operațiilor efectuate de programe, C++ permite acestora să utilizeze parametrii din linia de comandă. În lecția 39, „Înțelegerea și utilizarea polimorfismului", vom discuta despre polimorfism, care permite unui obiect să își schimbe forma în timpul rulării programului. Dar, înainte de a trece la lecția 39, asigurați-vă că ați reținut următoarele aspecte importante:

- Atunci când rulați un program de la promptul sistemului, informațiile pe care le tastați constituie linia de comandă a programului.
- Pentru a permite programelor să acceseze conținutul liniei de comandă, C++ transmite funcției *main* doi parametri, *argc* și *argv*.
- Parametrul *argc* conține numărul intrărilor din linia de comandă.
- Parametrul *argv* este un vector de pointeri la șiruri de caractere, fiecare dintre acestea conținând câte un parametru din linia de comandă.
- În funcție de compilator, este posibil ca programele să aibă acces la un al treilea parametru al funcției *main*, *env*, care este un vector de pointeri la șiruri de caractere ce conțin variabilele de mediu.

Lecția 39

Înțelegerea și utilizarea polimorfismului

Atunci când programatorii vorbesc despre C++ și programarea orientată spre obiect, un termen pe care îl vor „arunca" (folosi, dar nu neapărat înțelege) frecvent în discuție este acela de polimorfism. Vorbind în general, *polimorfismul* reprezintă capacitatea unui obiect de a-și modifica forma. Dacă analizăm cuvântul în sine, vom vedea că *poli* înseamnă multe, iar *morfism* se referă la schimbarea formei. Un obiect polimorf este, așadar, un obiect care poate asuma mai multe forme. Lecția de față prezintă polimorfismul și modul în care obiectele polimorfe pot fi utilizate în program pentru a simplifica și reduce codul. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- Polimorfismul este capacitatea unui obiect de a-și modifica forma în timpul rulării programului.
- Crearea de obiecte polimorfe în C++ este ușoară.
- Pentru crearea de obiecte polimorfe, programele trebuie să utilizeze funcții *virtuale*.
- O funcție *virtuală* este o funcție a unei clase de bază al cărei nume este precedat de cuvântul cheie *virtual*.
- Orice clasă derivată dintr-o clasă de bază poate să utilizeze sau să supradefinească o funcție *virtuală*.
- Pentru a crea un obiect polimorf veți utiliza un pointer la un obiect al clasei de bază.

Despre polimorfism

Un obiect polimorf este un obiect care poate avea diferite forme pe parcursul rulării programului. De exemplu, să presupunem că sunteți un programator care lucrează pentru compania de telefoane și trebuie să scrieți un program care simulează funcționarea unui telefon. Gândindu-vă la modurile în care oamenii folosesc telefonul, veți identifica rapid operații uzuale, precum formarea unui număr, sunatul telefonului, închiderea receptorului, semnalul de ocupat. Pe baza acestor operații veți defini clasa următoare:

```
class phone {
  public:
    void dial(char *number) { cout << "Dialing " <<
      number << endl; }
    void answer(void) { cout << "Waiting to answer call" <<
      endl; }
```

```
      void hangup(void) { cout << "Done with call-hanging up"
        << endl; }
      void ring(void) { cout << "Ring, ring, ring" << endl; }
      phone(char *number) { strcpy(phone::number, number); };
  private:
      char number[13];
  };
```

Programul următor, *PhoneOne.CPP*, utilizează clasa *phone* pentru a crea un obiect telefon:

```
#include <iostream.h>
#include <string.h>

class phone {
  public:
    void dial(char *number) { cout << "Dialing " <<
      number << endl; }
    void answer(void) { cout << "Waiting to answer call" <<
      endl; }
    void hangup(void) { cout << "Done with call-hanging up"
      << endl; }
    void ring(void) { cout << "Ring, ring, ring" << endl; }
    phone(char *number) { strcpy(phone::number, number); };
  private:
    char number[13];
  };

void main(void)
  {
    phone telephone("555-1212");
    telephone.dial("212-555-1212");
  }
```

Atunci când prezentați programul șefului, acesta vă spune că programul nu face deosebirea dintre telefoanele cu disc și cele cu butoane și că nu recunoaște nici telefoanele cu fise, care pot solicita utilizatorului introducerea unei monede înainte de efectuarea unui apel.

Deoarece cunoașteți conceptul de moștenire, vă hotărâți să derivați clasele *touch_tone* și *pay_phone* din clasa *phone*, ca mai jos:

```
class touch_tone : phone {
  public:
    void dial(char *number){ cout << "Beep Beep Dialing " <<
      number << endl;}
    touch_tone(char *number) : phone(number) { }
  };

class pay_phone : phone {
  public:
    void dial(char *number) { cout << "Please deposit " <<
                              amount << " cents" << endl;
                              cout << "Dialing " << number
                              << endl; }
    pay_phone(char *number, int amount) : phone(number) {
                              pay_phone::amount = amount; }
  private:
    int amount;
  };
```

După cum puteți vedea, clasele *touch_tone* și *pay_phone* definesc propriile metode *dial*. Dacă presupuneți că programul consideră metoda *dial* a clasei *phone* ca fiind corespunzătoare unui telefon cu disc, atunci nu mai este nevoie să creați o clasă distinctă pentru telefoanele cu disc. Programul următor, *NewPhone.CPP*, utilizează aceste clase pentru a crea obiectele *rotary*, *touch_tone* și *pay_phone*:

```
#include <iostream.h>
#include <string.h>

class phone {
  public:
    void dial(char *number) { cout << "Dialing " <<
      number << endl; }
    void answer(void) { cout << "Waiting to answer call" <<
      endl; }
    void hangup(void) { cout << "Done with call—hanging up"
      << endl; }
    void ring(void) { cout << "Ring, ring, ring" << endl; }
    phone(char *number) { strcpy(phone::number, number); };
```

```
   protected:
     char number[13];
   };

class touch_tone   phone {
   public:
     void dial(char *number){ cout << "Beep Beep Dialing " <<
       number << endl;}
     touch_tone(char *number) : phone(number) { }
   };

class pay_phone   phone {
   public:
     void dial(char *number) { cout << "Please deposit   <<
                                    amount <<   cents" << endl;
                                 cout << "Dialing   << number
                                    << endl; }
     pay_phone(char *number, int amount)    phone(number) {
                                   pay_phone::amount = amount; }
   private:
     int amount;
   };

void main(void)
   {
     phone rotary("303-555-1212");
     rotary.dial("602-555-1212");

     touch_tone telephone("555-1212");
     telephone.dial("212-555-1212");

     pay_phone city phone("555-1111", 25);
     city_phone.dial("212-555-1212");
}
```

După compilarea și rularea programului *NewPhone.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> NewPhone <Enter>
Dialing 602-555-1212
Beep Beep Dialing 212-555-1212
Please deposit 25 cents
Dialing 212-555-1212
```

Așa cum spuneam, un obiect polimorf este un obiect care își modifică forma pe durata rulării programului. Programul anterior, de exemplu, nu a folosit obiecte polimorfe. Cu alte cuvinte, nici un obiect nu și-a modificat forma.

Crearea unui obiect telefon polimorf

După ce arătați superiorului noul program pentru telefoane, acesta vă spune că obiectul telefon trebuie să poată simula, la cerere, un telefon cu disc, unul cu butoane sau unul cu fise. Cu alte cuvinte, obiectul telefon ar putea reprezenta la un apel un telefon cu butoane, la apelul următor un telefon cu fise, și tot așa. În acest fel, obiectul telefon și-ar putea modifica forma de la un apel la altul.

Singura funcție care diferă între clasele telefon pe care le-ați creat este metoda *dial*. Pentru a crea un obiect polimorf, veți declara mai întâi ca *virtuale* acele funcții ale clasei de bază care diferă de funcțiile corespunzătoare din clasele derivate, prin precedarea prototipurilor și antetelor funcțiilor respective cu cuvântul cheie *virtual*, așa cum este ilustrat aici:

```
class phone {
  public:
    virtual void dial(char *number) { cout << "Dialing " <<
      number << endl; }
    void answer(void) { cout << "Waiting to answer call" <<
      endl; }
    void hangup(void) { cout << "Done with call—hanging up"
      << endl; }
    void ring(void) { cout << "Ring, ring, ring" << endl; }
    phone(char *number) { strcpy(phone::number, number); };
  protected:
    char number[13];
};
```

În continuare, veți crea în program un pointer la un obiect al clasei de bază. În cazul programului cu telefoane, veți crea un pointer la clasa de bază *phone*, ca mai jos:

```
phone *poly_phone;
```

Pentru a modifica forma unui obiect, este suficient să atribuiți pointerului adresa obiectului unei clase derivate, ca aici:

```
poly_phone = (phone *) &home_phone;
```

Şirul *(phone *)* care urmează după operatorul de atribuire reprezintă o *conversie de tip*, care informează compilatorul de C++ că atribuirea adresei unei variabile de un anume tip *(touch_tone)* unui pointer la o variabilă de un alt tip *(phone)* este în regulă. Deoarece programul poate atribui pointerului la obiect *poly_phone* adresele diferitor obiecte, acest obiect poate avea mai multe forme, fiind aşadar polimorf. Următorul program, *PolyMorp.CPP*, foloseşte aceste tehnici pentru a crea un obiect telefon polimorf. Pe timpul rulării programului, obiectul *poly_phone* se schimbă de la un telefon cu disc la un telefon cu butoane şi la un telefon cu fise:

```
#include <iostream.h>
#include <string.h>

class phone {
  public:
    virtual void dial(char *number) { cout << "Dialing " <<
      number << endl; }
    void answer(void) { cout << "Waiting to answer call" <<
      endl; }
    void hangup(void) { cout << "Done with call—hanging up"
      << endl; }
    void ring(void) { cout << "Ring, ring, ring" << endl; }
    phone(char *number) { strcpy(phone::number, number); };
  protected:
    char number[13];
  };

class touch_tone : phone {
  public:
    void dial(char *number){ cout << "Beep Beep Dialing " <<
      number << endl;}
    touch_tone(char *number) : phone(number) { }
  };

class pay_phone : phone {
  public:
    void dial(char *number) { cout << "Please deposit " <<
                              amount << " cents" << endl;
                              cout << "Dialing " << number
                                 << endl; }
    pay_phone(char *number, int amount) : phone(number) {
```

```
                              pay_phone::amount = amount; }
  private:
    int amount;
  };

void main(void)
  {
    pay_phone city_phone("702-555-1212", 25);
    touch_tone home_phone("555-1212");
    phone rotary("201-555-1212");

    // Facem obiectul un telefon cu disc
    phone *poly_phone = &rotary;

    poly_phone->dial("818-555-1212");

    // Modificam forma obiectului la un telefon cu butoane
    poly_phone = (phone *) &home_phone;
    poly_phone->dial("303-555-1212");

    // Modificam forma obiectului la un telefon cu fise
    poly_phone = (phone *) &city_phone;
    poly_phone->dial("212-555-1212");
  }
```

După compilarea și rularea programului *PolyMorp.CPP*, pe ecran vor fi afișate următoarele:

```
C:\> PolyMorp <Enter>
Dialing 818-555-1212
Beep Beep Dialing 303-555-1212
Please deposit 25 cents
Dialing 212-555-1212
```

Deoarece obiectul *poly_phone* își modifică forma pe parcursul rulării programului, acest obiect este polimorf.

Obiectele polimorfe își pot schimba forma în timpul rulării programului



Un obiect polimorf este un obiect care își poate modifica forma pe durata rulării programului. Pentru crearea unui obiect polimorf, programul va utiliza un pointer la un obiect al clasei de bază. Programul va atribui, apoi, acestui pointer adresa unui obiect al unei clase derivate. De fiecare dată când programul atribuie pointerului un obiect al unei clase diferite, obiectul indicat de pointer (care este polimorf) își modifică forma. Programele construiesc obiectele polimorfe pe baza funcțiilor *virtuale* ale claselor de bază.

Despre funcțiile virtuale pure

Așa cum ați văzut, pentru crearea unui obiect polimorf, programele definesc una sau mai multe metode ale unei clase de bază ca funcții *virtuale*. Când derivați diferite clase, acestea pot oferi o funcție proprie care se execută în locul funcției *virtuale* din clasa de bază, sau pot folosi chiar acea funcție (cu alte cuvinte, nu definesc o metodă proprie). În funcție de program, pot exista situații în care nu are sens pentru clasa de bază să definească o funcție *virtuală*. Spre exemplu, tipurile obiectelor derivate ar putea fi atât de diferite încât nici unul dintre aceste obiecte nu va folosi metoda din clasa de bază. În asemenea cazuri, în locul definirii instrucțiunilor pentru funcția *virtuală* din clasa de bază, programele pot crea o *funcție virtuală pură* care nu conține nici o instrucțiune.

Pentru a crea o funcție virtuală pură, programul specifică prototipul funcției, dar nu și instrucțiunile acesteia. În schimb, programul atribuie funcției valoarea zero, așa cum se vede aici:

```
class phone {
  public:
    virtual void dial(char *number) = 0; // Functie virtuala
                                         // pura
    void answer(void) { cout << "Waiting to answer call" <<
      endl; }
    void hangup(void) { cout << "Done with call—hanging up"
      << endl; }
    void ring(void) { cout << "Ring, ring, ring" << endl; }
    phone(char *number) { strcpy(phone::number, number); };
  protected:
    char number[13];
};
```

Fiecare clasă derivată pe care o creați în program trebuie să definească o funcție pentru fiecare funcție *virtuală* pură din clasa de bază. Dacă o clasă derivată omite definirea funcției corespunzătoare unei funcții *virtuale* pure, atunci compilatorul de C++ va genera erori de sintaxă.

Ce trebuie să știți

Polimorfismul reprezintă capacitatea unui obiect de a-și modifica forma pe durata execuției programului. Lecția de față a studiat etapele ce trebuie parcurse pentru a crea obiecte polimorfe. În lecția 40, „Utilizarea excepțiilor din C++ pentru tratarea erorilor", veți învăța să folosiți excepțiile din C++ pentru a spori fiabilitatea programelor. Dar, înainte de a trece la lecția 40, asigurați-vă că ați reținut următoarele aspecte importante:

- ☑ Un obiect polimorf își poate modifica forma în timpul rulării programului.
- ☑ Obiectele polimorfe se creează prin utilizarea claselor derivate dintr-o clasă de bază existentă.
- ☑ În cadrul clasei de bază pentru un obiect polimorf, se definesc una sau mai multe funcții ca fiind *virtuale.*
- ☑ În general, obiectele polimorfe diferă prin modul de utilizare al funcțiilor *virtuale* din clasa de bază.
- ☑ Pentru crearea unui obiect polimorf trebuie creat un pointer la un obiect al clasei de bază.
- ☑ Schimbarea formei unui obiect polimorf, se face prin simpla indicare a unui alt obiect, atribuind pointerului la obiectul polimorf adresa noului obiect.
- ☑ O funcție *virtuală* pură este o funcție *virtuală* căreia clasa de bază nu îi definește instrucțiunile. În schimb, clasa de bază atribuie acestei funcții valoarea 0.
- ☑ Clasele derivate trebuie să conțină definiții de funcții pentru fiecare funcție virtuală *pură* din clasa de bază.

Lecția 40

Utilizarea excepțiilor din C++ pentru tratarea erorilor

După scrierea și depanarea (înlăturarea erorilor) unui număr mare de programe, veți începe să anticipați erorile de execuție care s-ar putea produce într-un program. De exemplu, dacă un program citește informații dintr-un fișier, acesta va trebui să testeze existența fișierului și dacă acel fișier poate fi deschis. De asemenea, dacă un program utilizează operatorul *new* pentru a aloca memorie, atunci trebuie testată și, eventual, tratată situația în care nu există memorie disponibilă. Pe măsură ce programele cresc în complexitate și în dimensiune, veți vedea că acestea vor conține un număr mare de astfel de teste. În lecția de față veți învăța să utilizați excepțiile din C++ pentru a simplifica testarea și tratarea erorilor din programe. Odată cu parcurgerea acestei lecții, veți înțelege următoarele aspecte cheie:

- O *excepție* reprezintă un eveniment neprevăzut, o eroare, care a survenit în program.
- În cadrul programului, excepțiile sunt definite sub formă de clase.
- Pentru a determina un program să urmărească apariția excepțiilor veți folosi instrucțiunea *try* din C++.
- Pentru a detecta o anumită excepție, programele utilizează instrucțiunea *catch* din C++.
- Pentru a genera o excepție la apariția unei erori, programele folosesc instrucțiunea *throw* din C++.
- La sesizarea (interceptarea) unei excepții, programul apelează o funcție specială (specifică excepției), care se numește *funcție de tratare a excepției.*
- Unele compilatoare (mai vechi) nu recunosc excepțiile din C++.

Notă: *Compilatorul **Turbo C++ Lite** oferit pe CD-ROM-ul care însoțește această carte nu recunoaște excepțiile din C++.*

C++ reprezintă excepțiile sub formă de clase

Scopul urmărit în utilizarea excepțiilor din C++ este simplificarea detecției și tratării erorilor în cadrul programelor. În mod ideal, atunci când întâlnesc o eroare neprevăzută (o excepție), programele ar trata eroarea într-un mod inteligent – prin contrast cu simpla încheiere a execuției. Fiecare excepție se definește într-un program sub forma unei clase. Spre exemplu, următoarele instrucțiuni definesc trei excepții legate de operațiile cu fișiere:

```
class file_open_error {};
class file_read_error {};
class file_write_error {};
```

Mai târziu, în această lecție, veți crea excepții care conțin variabile și funcții membru. Deocamdată, însă, rețineți că fiecare excepție corespunde unei clase.

Cum determinați C++ să detecteze excepții

Înainte ca programele să poată detecta și răspunde la o excepție, trebuie să utilizați instrucțiunea *try* din C++ pentru a activa detectarea erorilor. Următoarea instrucțiune *try*, de pildă, activează detectarea excepțiilor pentru apelul funcției *file_copy*:

```
try {
     file_copy("SOURCE.TXT", "TARGET.TXT");
};
```

Imediat după o instrucțiune *try*, programul trebuie să plaseze una sau mai multe instrucțiuni *catch* pentru a determina dacă a apărut vreo excepție și, dacă da, care anume, ca mai jos:

```
try {
     file_copy("SOURCE.TXT", "TARGET.TXT");
};

catch (file_open_error) {
   cerr << "Error opening the source or target file" << endl;
   exit(1);
}

catch (file_read_error) {
   cerr << "Error reading the source file" << endl;
   exit(1);
}

catch (file_write_error) {
   cerr << "Error writing to target file" << endl;
   exit(1);
}
```

După cum puteți vedea, codul testează erorile din operațiile cu fișiere pe care le-am definit anterior ca excepții. În acest exemplu, codul afișează pur și simplu un mesaj și apoi se încheie, indiferent de tipul erorii. În mod ideal, codul ar trebui să răspundă diferit, încercând, eventual, să elimine cauza erorii pentru a putea încerca din nou efectuarea operației. În cazul în care funcția se încheie cu succes și nu generează o excepție, C++ va ignora instrucțiunile *catch*.

Utilizarea instrucțiunii throw pentru generarea unei excepții

C++ nu generează, de fapt, excepții. În schimb, excepțiile sunt generate de program prin intermediul instrucțiunii *throw*. Spre exemplu, în cadrul funcției *file_copy*, programul poate testa și genera o excepție, ca mai jos:

```
void file_copy(char *source, char *target)
  {
  char line[256];
  ifstream input_file(source);
  ofstream output_file(target);
  if (input_file.fail())
    throw(file_open_error);
  else if (output_file.fail())
    throw(file_open_error);
  else
    {
      while ((! input_file.eof()) && (! input_file.fail()))
        {
          input_file.getline(line, sizeof(line));
          if (! input_file.fail())
            output_file << line << endl;
          else
            throw(file_read_error);
          if (output_file.fail())
            throw(file_write_error);
        }
    }
  }
```

Precum vedeți, programul folosește instrucțiunea *throw* pentru a genera excepțiile corespunzătoare.

Cum funcționează excepțiile



Atunci când utilizați excepții, programele testează apariția erorilor și, dacă este necesar, generează o excepție cu ajutorul instrucțiunii *throw*. Atunci când întâlnește o instrucțiune *throw*, C++ activează funcția corespunzătoare de tratare a excepției (o funcție ale cărei instrucțiuni se definesc în clasa excepției). După încheierea funcției de tratare a excepției, C++ cedează controlul primei instrucțiuni care urmează instrucțiunii *try* ce a activat detectarea excepțiilor. Apoi, prin intermediul instrucțiunilor *catch*, programul poate să determine excepția apărută și să răspundă în consecință.

Definirea unei funcții de tratare a excepției

Atunci când programul generează o excepție, C++ execută o funcție de tratare a excepției ale cărei instrucțiuni se definesc în clasa respectivei excepții. Spre exemplu, următoarea clasă de excepție, *nuke_meltdown*, definește instrucțiunile pentru tratarea excepției în funcția *nuke_meltdown*:

```
class nuke_meltdown {
  public:
    nuke_meltdown(void) { cerr << "\a\a\aRun! Run! Run!" <<
      endl; }
};
```

În acest exemplu, atunci când programul generează excepția *nuke_meltdown*, C++ execută instrucțiunile funcției *nuke_meltdown* și apoi cedează controlul primei instrucțiuni care urmează instrucțiunii *try* ce a activat detectarea excepțiilor. Următorul program, *MeltDown.CPP*, ilustrează modul de utilizare a funcției *nuke_meltdown*. Programul utilizează o instrucțiune *try* pentru a activa detectarea excepțiilor. Apoi este apelată funcția *add_u232* cu un parametru. Dacă valoarea parametrului este mai mică decât 255, funcția se va încheia cu succes. Dacă valoarea parametrului depășește, însă, 255, atunci funcția va genera excepția *nuke_meltdown*:

```
#include <iostream.h>

class nuke_meltdown {
  public:
```

```
      nuke_meltdown(void) { cerr << "\a\a\aRun! Run! Run!" <<
        endl; }
};

void add u232(int amount)
   {
      if (amount < 255)
         cout << "Amount of u232 is OK" << endl;
      else
         throw nuke_meltdown();
   }

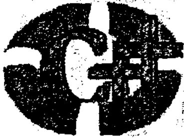
void main(void)
   {
      try {
         add_u232(255);
      }
   catch (nuke_meltdown) {
      cerr << "Run Faster" << endl;
   }
}
```

La compilarea şi rularea programului *MeltDown.CPP*, pe ecran vor fi afişate următoarele:

```
C:\> MeltDown <Enter>
Run! Run! Run!
Run Faster
```

Dacă urmăriţi codul sursă din program care generează fiecare mesaj, puteţi sesiza parcursul excepţiei prin funcţia de tratare şi înapoi la instrucţiunea *catch*. De exemplu, prima linie afişată este generată de funcţia *nuke_meltdown*. Apoi, instrucţiunea *catch* care detectează excepţia afişează pe ecran cea de a doua linie.

Definirea unei funcții de tratare a excepției



La detectarea unei excepții în cadrul programului, C++ execută o funcție specială numită funcție de tratare a excepției. Definirea unei funcții de tratare a excepției se face prin simpla creare în clasa excepției respective a unei funcții ce are același nume cu cel al excepției (asemeni unui constructor). Când programul generează apoi o excepție, C++ va apela automat funcția de tratare a excepției. În mod ideal, funcția de tratare a excepției ar trebui să efectueze operații care să ducă la remedierea erorii, astfel încât programul să poată relua operația care a cauzat excepția cu pricina. După încheierea funcției de tratare a excepției, execuția programului continuă cu prima instrucțiune care urmează instrucțiunii *try* ce a activat detectarea excepțiilor.

Utilizarea variabilelor membru ale unei excepții

În exemplele anterioare, programele puteau să determine, printr-o instrucțiune *catch*, excepția apărută și să răspundă în consecință. În mod ideal, cu cât programul poate obține mai multe informații despre o excepție, cu atât va putea să răspundă mai bine la eroarea apărută. Spre exemplu, în cazul unei excepții *file_open_error*, programul trebuie să cunoască numele fișierului care a cauzat eroarea. De asemenea, pentru excepțiile *file_read_error* și *file_write_error*, programul ar putea avea nevoie de poziția din fișier în care a apărut eroarea. Pentru a reține astfel de informații despre o excepție, programul nu trebuie decât să adauge variabile membru în clasa excepției respective. La generarea ulterioară a unei excepții, programul va transmite informațiile funcției de tratare sub formă de parametri, așa cum este ilustrat aici:

```
throw file_open_error(source);
throw file_read_error(344)
```

În funcția de tratare a excepției vor fi incluse instrucțiuni care atribuie parametrii variabilelor membru corespunzătoare ale clasei (foarte similar cu cazul unei funcții constructor). De exemplu, instrucțiunile următoare modifică excepția *file_open_error*, pentru a atribui numele fișierului care a cauzat eroarea variabilei membru corespunzătoare:

```
class file_open_error {
  public:
    file_open_error(char *filename) {
      strcpy(file_open_error::filename, filename); }
    char filename[255]
};
```

Tratarea excepțiilor neprevăzute

În lecția 12, „Utilizarea bibliotecilor de execuție", ați învățat despre biblioteca de funcții pentru execuție pe care o oferă compilatoarele de C++ și care poate fi utilizată în cadrul programelor. Pe măsură ce parcurgeți documentația pentru aceste funcții, este posibil să întâlniți funcții care generează excepții specifice. În asemenea cazuri, ar trebui să testați în programe apariția excepțiilor corespunzătoare. În mod implicit, la generarea unei excepții ce nu poate fi interceptată (când programul nu dispune de o funcție de tratare a excepției corespunzătoare), programul execută o funcție de tratare implicită, oferită de C++. De cele mai multe ori, funcția de tratare implicită va încheia execuția programului. Următorul program, *UnCaught.CPP*, ilustrează modul în care funcția implicită de tratare a excepției încheie execuția programului:

```
#include <iostream.h>

class some_exception { };

void main(void)
  {
     cout << "About to throw exception" << endl;
     throw some_exception();
     cout << "Exception thrown" << endl;
  }
```

În exemplul de mai sus, generarea de către program a excepției (neinterceptate în cod) face ca C++ să invoce funcția implicită de tratare a excepției, iar aceasta încheie execuția programului. Acesta este motivul pentru care ultima instrucțiune din program, care afișează mesajul despre excepția generată, nu se execută niciodată. În locul utilizării funcției de tratare implicite din C++, programele își pot defini propria funcție implicită de tratare a excepțiilor. Pentru a informa C++ despre funcția de tratare implicită din program, codul apelează la funcția *set_unexpected* din biblioteca de execuție. Prototipul funcției *set_unexpected* este definit în fișierul de antet *except.h*.

Specificarea excepțiilor ce pot fi generate de o funcție

După cum ați învățat, prototipul unei funcții vă permite să definiți tipul întors de acea funcție și tipurile parametrilor acceptați. Atunci când folosiți excepții în programe, puteți utiliza prototipul unei funcții pentru a specifica și excepțiile pe care acea funcție le poate genera. Spre exemplu, următorul prototip de funcție informează compilatorul că funcția *power_plant* poate genera excepțiile *melt_down* și *radiation_leak*:

```
void power_plant(long power_needed) throw (melt_down,
  radiation_leak);
```

Prin această precizare a excepțiilor posibile în prototipul funcției, alți programatori care citesc codul pot afla imediat ce excepții vor trebui să urmărească atunci când utilizează funcția respectivă.

Excepții și clase

Atunci când creați clase, este posibil să doriți definirea de excepții specifice unei clase. Crearea unei excepții specifice unei clase se face prin simpla includere a excepției ca unul dintre membrii *publici* ai clasei respective. De exemplu, următoarea definiție a clasei *string_class* declară două excepții:

```
class string {
  public:
    string(char *str);
    void fill_string(*str);
    void show_string(void);
    int string_length(void)
    class string_empty { }
    class string_overflow { }
  private:
    int length;
    char string[255];
}
```

După cum puteți observa, clasa *string* definește excepțiile *string_empty* și *string_overflow*. În cadrul programului, puteți testa apariția unei excepții de clasă utilizând operatorul de rezoluție globală și numele clasei, ca aici:

```
try {
  some_string.fill_string(some_long_string)
};

catch (string::string_overflow) {
  cerr << "String length exceeded-characters truncated" <<
     endl;
}
```

Ce trebuie să știți

Scopul excepțiilor este să simplifice și să îmbunătățească facilitățile de detecție și de tratare a erorilor din programe. Pentru testarea și detectarea excepțiilor, programele folosesc instrucțiunile: *try*, *catch* și *throw*. Cunoștințele despre excepții vă vor completa cunoștințele de programare în C++. Dar, înainte de a continua cu programarea în C++, asigurați-vă că ați reținut următoarele aspecte importante:

☑ O excepție este o eroare neprevăzută care apare în program.

☑ Programele ar trebui să detecteze și să răspundă la (să trateze) excepții.

☑ În cadrul programelor, fiecare excepție este definită sub forma unei clase.

☑ Folosiți instrucțiunea *try* pentru a determina compilatorul de C++ să activeze detectarea excepțiilor.

☑ După instrucțiunea *try*, ar trebui să plasați imediat instrucțiuni *catch* pentru a determina dacă a apărut vreo excepție și care este aceasta.

☑ C++, în sine, nu generează excepții. De fapt, excepțiile sunt generate de programe prin intermediul instrucțiunii *throw*.

☑ Atunci când interceptează o excepție, programul apelează o funcție specială numită funcție de tratare a excepției.

☑ Atunci când programele utilizează excepții, puteți specifica în prototipul unei funcții excepțiile pe care aceasta le poate genera.

☑ La utilizarea funcțiilor din biblioteca de execuție, țineți cont de faptul că unele funcții ar putea genera excepții.

☑ Dacă programul generează o excepție care nu este interceptată, C++ va invoca o funcție de tratare implicită.

☑ Fișierul de antet *except.h* specifică prototipurile de funcții pe care programele le pot utiliza pentru a defini propriile funcții implicite de tratare a excepțiilor și încheiere a programului.

Index